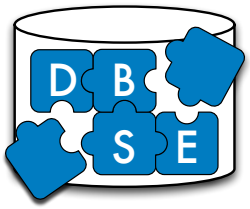
University of Magdeburg

School of Computer Science



Databases
and
Software
Engineering

Master Thesis

On the Impact of Hardware on Relational Join Processing

Author:

David Broneske

August 19, 2013

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake
Dipl.-Inform. Martin Schäler
Dipl.-Inform. Thomas Thüm

Department of Technical & Business Information Systems

Broneske, David:
On the Impact of Hardware on Relational Join Processing
Master Thesis, University of Magdeburg, 2013.

Abstract

Current database systems are confronted with increasing demands considering processed data and expected response time. Hence, given hardware has to be fully-exploited to fulfill the requirements. To this end, modern hardware represents an opportunity to accelerate database operations. However, extensive evaluations of database operation performance under changing hardware have not been accomplished to the best of our knowledge. In our work, we derive hypotheses on algorithm performance regarding the used hardware from literature and present possible impact factors. With this, we implement an evaluation framework to test our hypotheses and evaluate the performance of join algorithms for different storage devices.

Acknowledgements

At first, I would like to thank Gunter Saake for his encouraging work. His contribution finally led me to the topic of this thesis. Special thanks to Martin Schäler for his continuous motivation and inspiration, which significantly improved the contribution of this work and helped to keep my thoughts in this thesis consistent. In addition, I am very thankful for having Thomas Thüm as advisor, because of his valuable comments on the content and style of this thesis. An important role considering my implementation played Sebastian Breß, who guided me through the internals of database management systems and C++-specific implementation threats.

Furthermore, I would like to thank Kai-Uwe Sattler for admitting to be me external reviewer and all those who helped my with inspiring talks, namely Norbert and Janet Siegmund, Veit Köppen, Wolfram Fenske, Reimar Schröter, Thomas Leich, and Benny Hapke.

Finally, a heartily thanks goes to my family and my friends who continuously supported me and were very tolerant for my limited time during this work.

Contents

List of Figures

List of Tables

List of Code Listings

List of Algorithms

1. Introduction

Database technology and demands on it has evolved immensely in the last decades. According to Larson [Lar13], commercial database management systems were originally designed for *online transactional processing (OLTP)* workloads on disk resident data that were executed on slow processors with scarce main memory. In contrast, today's workload include OLTP as well as OLAP (online analytical processing) and processing capabilities have improved to strong multi-core CPUs and plenty available RAM, so that most of the data can be held in the in-memory buffer pool. As a consequence, traditional database management systems have to be adapted to the new workloads and hardware capabilities.

The force to increase the processing capability of databases is even higher considering diverse scenarios reaching from data management for embedded systems [Lei12] to large-scale applications. As a consequence, processing capabilities have to be fully-exploited and new hardware trends have to be examined to provide a high performance in the future. Considering new hardware possibilities, on the one hand, new hardware includes new co-processing devices, such as *graphics processing units (GPUs)* and *field-programmable gate arrays (FPGAs)*. Especially GPUs as co-processing devices for database systems have been pushed into sight in the last decade [GLW+04, FHL+07, HYF+08, KLMV12], making co-processing a popular technique for performance gains and should be continued for FPGAs. On the other hand, storage configurations that rely on the use of SSDs and huge amounts of RAM are at hand accelerating access to processed data.

Arising from the usage of new hardware, we argue that database management systems have to be tailored to the new hardware to exploit its processing power. This statement is underlined by Stonebraker et al. claiming that "it's time for a complete rewrite" [SMA+07]. Consequently, an important task is to examine processing capabilities of the new hardware and map these results to the algorithms of our database operations. In literature, there are numerous different algorithms proposed for the most

common database operations. However, their evaluations are done on a single machine using a predefined hardware configuration and, thus, present no differences of the algorithms with respect to different hardware devices. Consequently, we want to initiate a first step towards a comparison of different *database operation implementations* under changing hardware with this work.

Goal of this Thesis

The goal of this thesis is to review and evaluate how the performance of database operations changes under varying hardware configurations. To this end, we provide the following contributions to reach a comprehensive evaluation of database operation performance.

1. Based on our literature review of publications of the main conferences considering database operations on different storage and processing devices, we derive assumptions to the performance of database operations under different hardware configurations.
2. Furthermore, we extract impact factors from literature that influence the performance of database operations and present identified impact factors in a well-structured way using feature models. This also helps us to describe visible relations between impact factors of our model.
3. Since the amount of impact factors is too huge to evaluate, because of continues parameter values, we contribute a reasonable limitation of the impact factors.
4. Considering found impact factors, we review different implementation techniques to provide variability in a framework helping us to evaluate the performance of database operations under changing impact factors.
5. We contribute an analysis of the qualitative and quantitative differences between presented implementation techniques.
6. Finally, we evaluate the performance of join algorithms under different storage configurations and present our results.

Structure of the Thesis

This thesis is structured as follows. We start with an extensive background chapter that introduces necessary topics to understand the following discussion about database operations and performance impacts. In Chapter 3, we present hardware that is currently used in the processing of database queries and also introduce current research which focuses on the use of new hardware for executing database operations. This chapter also reviews comparisons of different algorithms for new hardware and performance evaluations in literature that are strongly related to our results. From this literature

review, we extract important impact factors on the performance of database operations including the impact of new hardware and summarize these factors in Chapter 4.

Considering changing hardware, we focus on different storage devices and their influence on the performance of database operations. To this end, we explain our evaluation method and environment in Chapter 6 and present important results of our evaluation in the following section. We end this thesis with a conclusion and present steps that have to be done in future work to continue the work of this thesis.

2. Background

In the following, we present basic background required for an unbiased analysis of join algorithms in a database system. A database system is a complex system and, thus, its core components have to be known in particular to state their impact on performance. Hence, we will present the classical system architecture of database systems in this chapter. Since join behavior in these systems is well known and documented, it will give us a good base for formulating performance assumptions on different systems. As an important obstacle in this architecture, we explain the access gap in the second section and finally present the four categories of join algorithms in Section 2.3.

2.1 Classical Database Architecture

The classical architectural model of a database has been studied in the eighties [BFJ+86] and three important models arose, which describe the components of a database system in a more or less fine granularity. These three models are: the *three-schema logical database architecture* [BFJ+86], the *ANSI/SPARC Reference Model* [BFJ+86, SSH11] and the *five-level schema architecture* [Här87].

The three-schema logical database architecture is a coarse classification of the system components into three schemas, namely the external, conceptual and internal schema. The ANSI/SPARC Reference Model takes up these three schemas and refines each of them with specific components as well as functionality that have to be provided. However, a good implementation-oriented view on the database is not given in these models.

For our work, we concentrate on the five-level schema architecture which is also an extension of the three-schema logical database architecture. Since the five-level schema architecture represents an implementation-oriented view on the database components [Här87], it is a useful guideline for our implementation and we present the five-level schema architecture in the following section in detail.

2.1.1 Five-Level Schema Architecture

The five-level schema architecture was introduced by Härder [Här87] in 1987 and is a hierarchical composition of the following five components: *data system*, *access system*, *storage system*, *buffer manager*, and *operating system*. Between components, there are defined interfaces, namely *set-oriented interface*, *record-oriented interface*, *internal record interface*, *system buffer interface*, *file interface*, and *device interface*. Relations between components and interfaces are visualized in Figure 2.1.

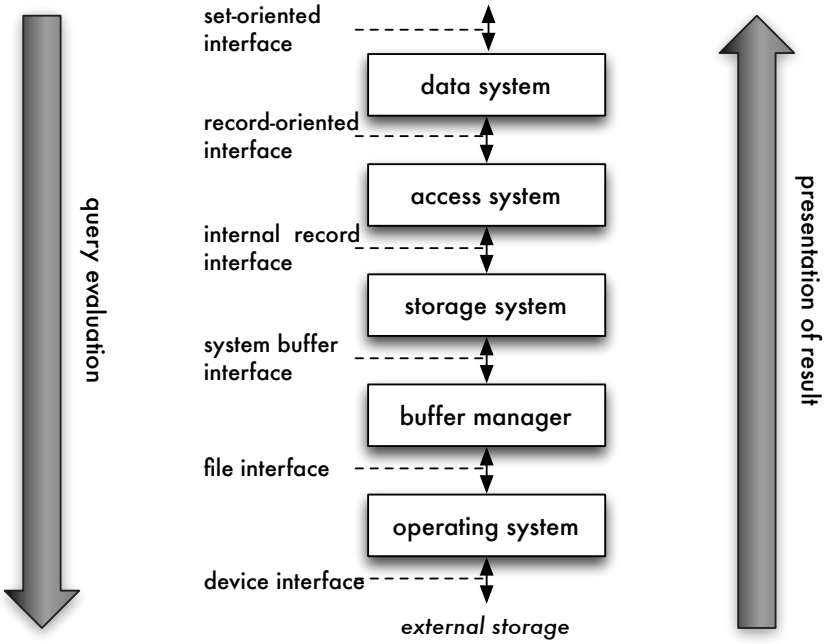


Figure 2.1: Five-level schema architecture – adapted from [SSH11].

The five-level schema architecture is a hierarchical model, consisting of five components, four interfaces between them, and two interface to external devices. The benefit of the five-level architecture is the usage of abstraction. Each component has its own degree of abstraction to the preceding and following component. For the purpose of data exchange between two connected components, there has to be a common interface between them [Här87].

The query processing starts at the top level where the query is described in a declarative query language, such as *SQL*. SQL is similar to the human language and allows a high level of abstraction [Här87]. The actual data required for answering a query is present on external storage devices, so the query passes every component downwards the hierarchy and is transformed step by step until the result data is retrieved from disk. After that, the data is passed upwards the hierarchy and transformed into the table-like representation. For detailed information on the query representation in the components and provided functionality, we review each component in the following.

2.1.2 Data System

The data system component is the most abstract component in the architecture and thus, the data representation is on the highest abstraction level. Data is accessed by a declarative query language, such as SQL, which is passed over the set-oriented interface to the data system.

The data system translates received queries and optimizes the execution order of operations. Therefore, knowledge about index structures and tables has to be present. Furthermore, it manages whether a user is permitted to read, update, or delete data of the specified tables (*access control*) and whether there are any integrity constraints violated by the query (*integrity control*) [SSH11].

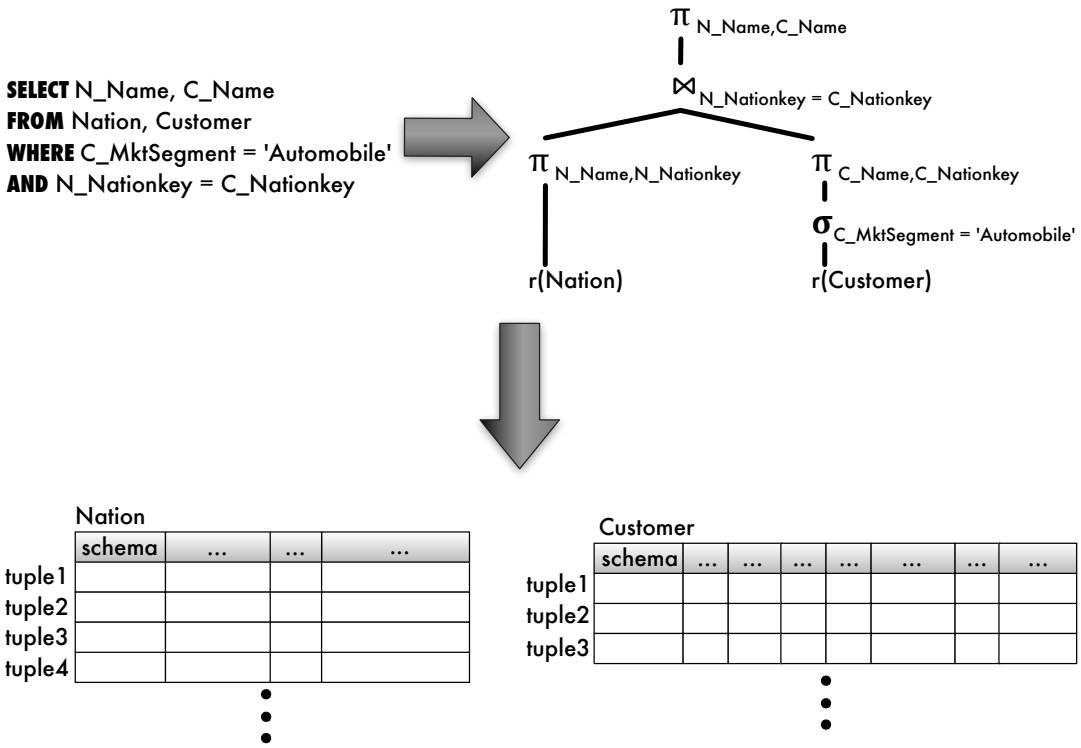


Figure 2.2: Query translation in the data system.

In Figure 2.2, we depict an example taken from the TPC-H benchmark schema[1] where the `nation` and `customer` table is joined and the output should only contain those records where the customer is in the automobile market segment. This SQL-Statement is translated into an optimized query plan, which we represent in relational algebra. The notation `r(x)` denotes that the relation `x` is taken and forwarded to the next operand. π is a projection, which puts out the input relation reduced to given attributes. The selection operand, which is represented as σ, puts out a relation with reduced rows which all have to fulfill the given selection condition. The binary operation $\bowtie$ denotes

[1] http://www.tpc.org/tpch/

the join operation, which takes two relations and combines their rows with the same value in the specified attributes. Further operations defined in the relational algebra are explained in [SSH13]. After optimization, the query plan is executed on the tables in the access system. For the communication, a record-oriented view on the tables is forwarded, which means we can navigate through the records of each table.

2.1.3 Access System

The access system has to provide the transformation of data from the table-wise view of the record-oriented interface to a tuple-oriented view of the internal record interface. Such an internal record is a uniform representation of tuples without considering its corresponding table. Hence, a uniform management of tuples independent of the source relation is provided. We depict the transformation in Figure 2.3, where the input tables are transformed to a list of tuples that can be processed.

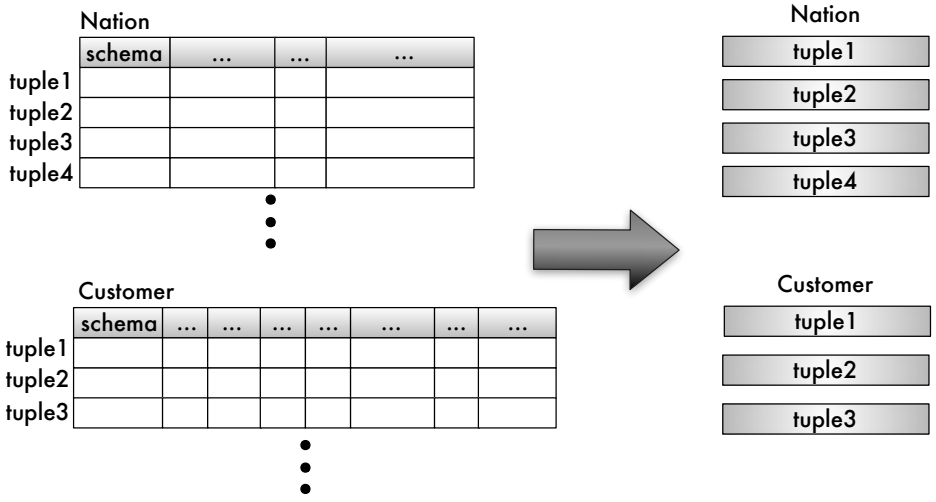


Figure 2.3: From table-wise to tuple-wise execution.

The transformation to an internal representation of tuples is necessary, because tuples are stored on pages written to disk in a binary format. Thus, there has to be a system which bridges the gap between binary representation of tuples and typed representation of tuples, namely, the access system.

Further typical tasks of the access system are to update index structures and the data dictionary. The data dictionary holds necessary information of all tables and is needed for typing the internal record to confirm to the record-oriented interface. Additionally, algorithms for sorting the output relations are implemented in the access system [Här87].

2.1.4 Storage System

The tuples that are processed in the access system are organized on pages. A page is an abstract container with a concrete size of bytes. Since tuples may be updated, a sequential storage of its tuples on pages is not possible in every scenario. Consequently,

there has to be a concept for identifying the storage location of tuples on pages. For the mapping, the storage system uses the *tuple identifier (TID) concept* which is also explained by Saake et al. [SSH11]. To identify the storage location of a tuple, the TID consists of two values that are the page number and an offset in the header of the page. The value behind the offset in the header points to the storage location of the tuple in the page.

An advantage of the TID concept is that a TID can also be used in index structures to reference tuples. Hence, it is possible to retrieve a specified tuple from a page by its TID found in the index structure. Moreover, if a new tuple is inserted, the storage system has to update the index structure as well.

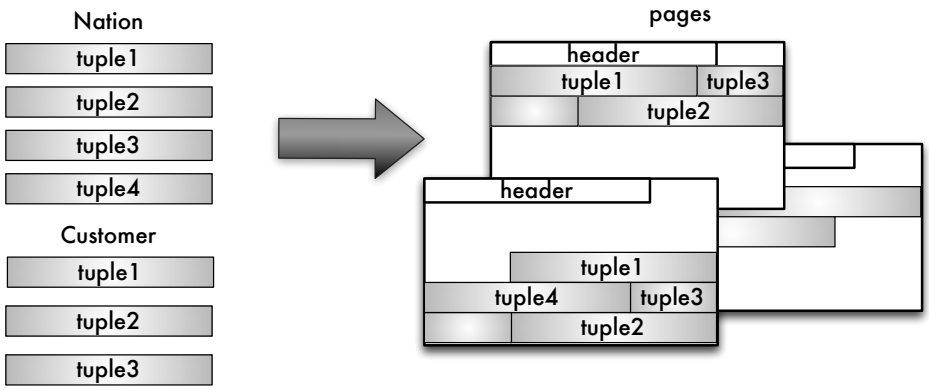


Figure 2.4: Tuples used in the access system are stored on pages.

When retrieving a tuple, the ID of the page has to be calculated according to the given TID. The storage location of the tuple on the page is typically stored in the header of the page. Pages that have to be processed by the storage system, e.g., those used in Figure 2.4, have to be loaded by the buffer system.

2.1.5 Buffer System

The buffer system provides pages requested by the storage system. Since in many scenarios the RAM is much smaller than the amount of data, all pages of a relation do not fit en bloc into main memory. Consequently, an efficient page replacement strategy is required and has to be implemented in the buffer system.

Easy ways of implementing a page replacement strategy are to replace the oldest page (*FIFO*) or the least recently used one (*LRU*). However, these strategies are not efficient in the context of database operations [SSH11]. Hence, numerous page replacement strategies were developed to overcome this issue. For a collection of important strategies, we refer to [SSH11].

First, when the storage system requests a page, it has to be searched for the page in the buffer pool. In the best case, the page is still cached and can be returned. However, if the page is not found, it has to be loaded from disk and is cached in the buffer pool,

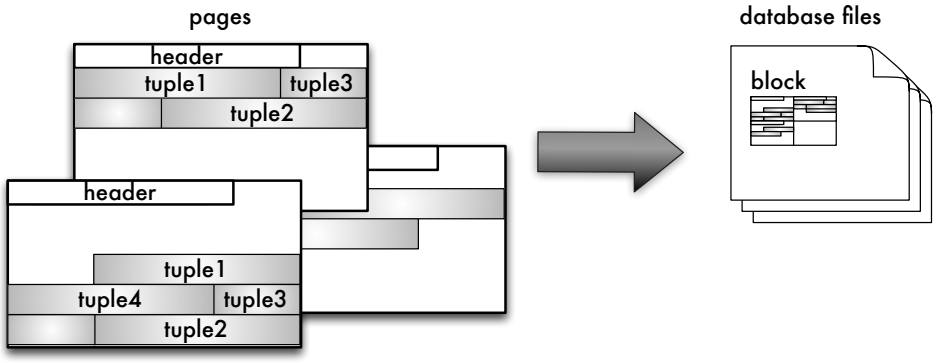


Figure 2.5: Load needed page from files.

if there is still enough space. When a requested page does not fit into the buffer pool anymore, a cached page has to be chosen to be replaced. After that, the page can be loaded into the buffer pool and is then accessible in the storage system. When loading the page, page data has to be extracted from files storing information in blocks, which we depict in Figure 2.5. Files represent the abstraction in the file interface which is the input of the operating system.

2.1.6 Operating System

The operating system is the lowest component in the five-level schema architecture. Its task is to provide mechanisms to read data from disk and write data to disk. For this, navigation on the disk has to be implemented to update files on disk, as depicted in Figure 2.6.

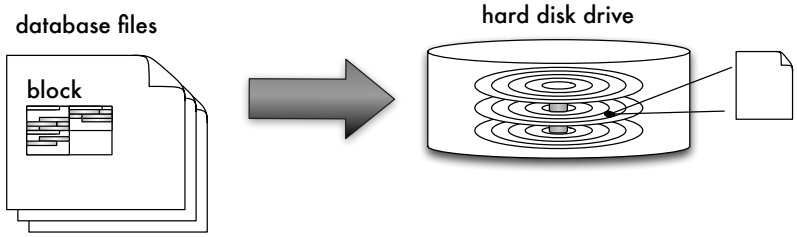


Figure 2.6: Extracting files from disk.

The operations implemented in the operating system are used to bring data from disk to main memory. Nevertheless, a high amount of disk accesses should be avoided, because the so-called *access gap* decreases performance. Hence, we review the access gap in the following section.

2.2 Access Gap

On the way through the processing of a query, data has to pass several different storage devices. The classification of the storage devices depends on access time and

costs [PSG08]. Since large amounts of fast storage devices are often too expensive, we have to cope with issues arising from limited space. Different levels of storage devices are classified in the memory hierarchy, which we present in the following section.

2.2.1 Memory Hierarchy

The memory hierarchy consists of three different groups, as we depict in Figure 2.7. The groups range from primary over secondary to tertiary memory. In general, memory located in a higher levels of the hierarchy offers higher access speeds, but also cost per MB increase the higher the device is classified in this hierarchy. As a consequence, in most of the systems there is a limited memory capacity of devices in the primary memory while devices in the tertiary memory offer plenty of space.

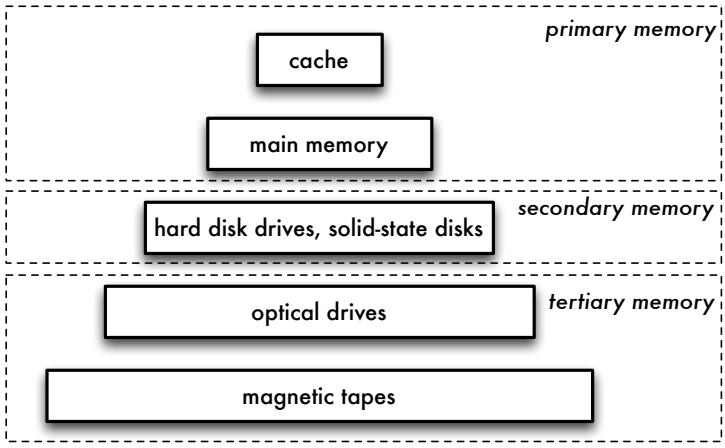


Figure 2.7: Memory hierarchy – adapted from [SSH11].

Since data processing typically follows specific access patterns, the principle of the *locality of reference* is considerable for achieving performance improvements. On the one hand, locality of reference means that data is used more than once in our processing (*temporal locality*). On the other hand, if a memory location is referenced in an operation, it is likely that nearby locations will be referenced as well (*spatial locality*) [SSH11]. To exploit the locality of reference, frequently accessed data is cached on memory devices situated in higher levels of the hierarchy while less often used data is stored on lower levels. Having only the most likely used data cached is especially important for the primary memory.

Primary Memory

The primary memory can be classified into cache and main memory [SSH11]. While cache memory consists of fast *static random-access memory* (*SRAM*), for main memory *dynamic random-access memory* is used which is slower, but less expensive. In general, both memory devices are volatile, which means that they require electricity to keep data stored. In case of a loss of electricity, stored data is lost. Consequently, persistent storage cannot rely on RAM memory devices.

In modern computers, there are several levels of cache memory, typically ranging from L1 to L3 cache. The L1 cache is the smallest, but is located on the CPU chip to grant fast access [BMK99]. The sizes of modern caches range from several kilobytes in L1 cache over several hundred kilobytes in L2 cache up to several megabytes in L3 cache. The highest amount of storage space in primary memory is the main memory which is nowadays several gigabytes big. Data that is not present in RAM has to be loaded from disks of the secondary memory. For detailed properties of current commodity hardware, we present some interesting numbers in Table 2.1.

Storage Module	Size	Latency	Transfer Rate
L1-Cache	32KB per core	$1.2ns$	$49GB/s$
L2-Cache	256KB	$3.2ns$	$32GB/s$
L3-Cache	6-8KB	$5.2ns$	$18GB/s$
DDR3 RAM	16-64GB	$11.25ns$	$12.8 - 25.6GB/s$
HDD	1TB	$8.5ms$	$300MB/s$

Table 2.1: Size, latency, transfer rates of current commodity hardware. Numbers taken from different sources[2] with an Intel Core i7-965 Extreme Edition and RAM of 1600MHz frequency and CL9-9-9.

Secondary Memory

The secondary memory includes only *non-volatile* storage devices, which means data is stored permanently even without electricity. Hence, secondary memory is an important storage location for databases to ensure persistency. Typical storage devices of secondary memory represent hard disk drives (*HDD*), which have a capacity of several hundreds of gigabytes up to a few terabytes.

Nevertheless, data in secondary memory cannot be directly processed and has to be loaded into the primary memory [SSH11]. Furthermore, the granularity of access is much higher on an HDD than on devices of the primary memory. For example, on an HDD, addressable blocks are often about 512 bytes big while in RAM every byte is addressable [SSH11].

An HDD consists of multiple spinning platters where data is stored in concentric tracks. To read or write, the heads have to be positioned to the local storage location. The disk access time depends on the seek time, rotational latency and transfer time [Den11]. Small random accesses cause high seek time and dominate the access time and, hence, a sequential access is favored for HDDs. In addition, access times are much higher on HDDs than on RAM, typically with a factor of 10^5 [SSH11]. This phenomenon is called the access gap, which we review in Section 2.2.

[2] http://en.wikipedia.org/wiki/Hard_disk_drive,
http://en.wikipedia.org/wiki/DDR3_SDRAM,
http://www.xbitlabs.com/articles/cpu/display/intel-core-i7_7.html,
http://www.alternate.de/Seagate/Seagate+ST1000DM003_1_TB,_Festplatte/html/product/963366/?

Tertiary Memory

Similar to secondary memory, the tertiary memory is a non-volatile storage device. However, in contrast to secondary memory, a tertiary memory device can be exchanged while the system is running. Furthermore, they offer a high amount of storage capacity at low cost, but with high access times. Typical tertiary storage devices are optical drives or magnetic tapes.

In databases, the tertiary memory is used for storing backups, because storing data on tertiary memory for operational access would mean to accept high delay which is not in the sense of databases. Nevertheless, they are a perfect medium for storing backup data of the database, because of their high capacity and exchangeability.

2.2.2 Access Gap Between HDD and RAM

Access time of the devices of different memory levels differ significantly. Especially, the gap between hard disk drives and random access memory are very high, which is also evident from Table 2.1. Saake et al. state that hard disk drives increase their storage space by about 70% a year, but access time only improves by 7% [SSH11]. In contrast, processing power of CPUs increased by 70% a year and access time of RAM by 50% a year [BMK99]. The high difference in access time between RAM and hard disk is called the access gap [SGG+99, BMK99, SSH11].

The significant deficit between performance and access time causes many operations to slow down their processing, because data is not available when needed. Thus, access to the secondary memory becomes the bottle neck of the system. To overcome this gap, there have to be either good algorithms or new hardware. We review a possible solution represented by *solid-state disks (SSD)* in Section 3.3.

Especially the access gap causes difficulties in designing suitable algorithms for database operations. For instance, join algorithms working on tables with sizes much bigger than available RAM have to be implemented considering the access gap. For this, we will present basic join algorithms in the following.

2.3 Relational Join Strategies

In this section, we present different join strategies introduced in literature. A relational join is a binary operation on two relations leading to a combination of them. In the following section, we start by introducing the Cartesian product as the basis of combination of relations and advancing to more restrictive methods.

2.3.1 Theory of Relational Joins

Relational joins are extensively reviewed in literature [Got75, Sha86, ME92]. The necessity for having joins arises, for instance, from the use of normal forms. Due to a favorable non-redundant storage, relations are split up to satisfy conditions for normal forms [SSH13]. However, if a query has to retrieve the whole relation, split relation

have to be reunited. In general, considering a multi-attribute key, even a multi-attribute foreign key would be created when splitting. Joining these tables would lead to a comparison in more than one attribute. Nevertheless, for simplicity, we only consider keys consisting of one attribute.

Cartesian Product

The theory of combining two relations is based on the *Cartesian product*. The Cartesian product results in a relation where each tuple of the first relation is combined with each tuple of the second relation [ME92]. Considering n tuples for the first relation and m tuples for the second relation, the cardinality of the result is $(n * m)$. Furthermore, the attributes of the resulting relation are the concatenation of attributes from both input relations.

The Cartesian product produces a combination of two relations, but this operation is not meaningful for reconstructing a relation which was split in the sense of normalization, since it does not combine only those tuples belonging to each other.

Theta Join

A more meaningful way of combining two relations is the *theta join*. The theta join can be represented by using a Cartesian product and a selection [ME92] while the selection checks if a condition between two join attribute values is satisfied. There are several join conditions possible in a theta join, for example: $=$, $\neq$, $<$, $>$, $\leq$, $\geq$.

The theta join as an extension to the Cartesian product has at most $(n * m)$ tuples in the result relation, but in most cases the cardinality of the result is much smaller. Furthermore, the set of attributes is the concatenation of attributes from both input tables.

An important join condition of the theta join is the "=", which is also known as *equi join*. Equi joins are especially considerable since they reunite two relations that were split because of normalization. The exact reconstruction of split relations using an equi join is called *natural join*.

Natural Join

The natural join as the inverse operation to the splitting of a relation because of normalization has to ensure that the original table with all original tuples and attributes is reconstructed. Since the splitting leads to an additional equally-named attribute in one of the resulting relations (representing the *foreign-key relationship*), this attribute has to be omitted in the result relation. Thus, in addition to the equi join execution, a projection on the specified attributes has to be done as well [ME92]. The projection is useful to reduce the storage cost of the join result and, furthermore, it does not lose any necessary information, because the normalization introduced this attribute.

There are several other joins, such as semi join, outer join, self join, which will not be covered here, as they are special cases of the natural join or theta join. For our further

course of action, we limit our consideration to the execution of natural joins, because of its memory saving.

In the next section, we review different ways of executing a natural join on two relations and present some exemplary algorithms clarifying these approaches. For this, we consider only ad-hoc-join algorithms to show basic concepts of join strategies [HCLS93]. Of course, joins differ when index structures can be used or even join indexes were created, but this is not in the scope of this work.

2.3.2 Nested-Loops Join

The easiest way of implementing a join is the nested-loops join [ME92]. As the name suggests, the nested-loops join consists of two or more loops that are nested into each other. In particular, considering two relations the first tuple of the outer relation is read and compared to every tuple in the inner relation. After that, the next tuple of the outer relation is read and again compared to every tuple in the inner relation.

```
1  Input: Relation A, JoinAttribute jA, Relation B, JoinAttribute jB
2  Result: Relation C
3  foreach Tuple a in A do
4      foreach Tuple b in B do
5          if a.jA = b.jB then
6              // keys match → join tuples
7              c:= join(a,b);
8              C.insert(c);
9          end
10     end
11 end
12 return C;
```

Algorithm 2.1: Nested-loops-join algorithm – adapted from [ME92].

In Algorithm 2.1, we show an exemplary algorithm for a nested-loops join. Since each tuple of relation `A` (with cardinality n) is compared to each tuple of relation `B` (with cardinality m), the nested-loops join has the computational complexity $\mathcal{O}(n*m)$. Hence, the nested-loops join is the *brute-force* strategy for a join and, thus, a good starting point for comparisons.

Because of its high complexity, the nested-loops join is hardly applicable for large relations. Nevertheless, it holds great potential regarding parallel execution which makes it still considerable especially for new hardware [ME92].

2.3.3 Block-Nested-Loops Join

An improvement of the nested-loops join reducing I/O-costs is the *block-nested-loops join*. Especially when considering the access gap (cf. Section 2.2), it is a promising way

to design algorithms to have a better I/O-behavior. Considering the nested-loops join, it is more efficient to take advantage of the underlying hard disk by aggregating tuples that are processed into chunks of tuples, the so-called blocks. We provide exemplary code in Algorithm 2.2 for the block-nested-loops-join.

```
1  Input: Relation A, JoinAttribute jA, Relation B, JoinAttribute jB
2  Result: Relation C
3  // Load as many pages as possible
4  foreach Block aBlock in A do
5      // Load page by page → block size = page size
6      foreach Block bBlock in B do
7          foreach Tuple a in aBlock do
8              foreach Tuple b in bBlock do
9                  if a.jA = b.jB then
10                     // Keys match → join tuples
11                     c:= join(a,b);
12                     C.insert(c);
13                 end
14             end
15         end
16     end
17 end
18 return C;
```

Algorithm 2.2: Algorithm of block-nested-loops join – adapted from [SSH11].

The size of blocks depends on the amount of available main memory. The outer relation `A` should take as much space as possible, while the inner relation `B` can be read page by page for comparison [ME92]. With this algorithm, already fetched pages can be utilized more efficiently. For additional improvements, the number of tuples in both relations has to be considered. The relation of the outer loop should be the smaller one, because most of the page misses occur in the outer loop and tuples of the inner loop are sequentially read.

The computational complexity of the block-nested-loops join is still $\mathcal{O}(n * m)$, because still each tuples of one relation is compared to each tuples of the other relation. One way to avoid some of the unnecessary comparisons represents the *hash join*.

2.3.4 Hash Join

The hash join is executed in two phases [ME92, SSH11]. In the first phase (the *build phase*), a *hash table* is build up on all tuples of one relation. For this, the join attribute of each tuple is hashed with a given *hash function* to determine the bucket the tuple is inserted into. Here, it is possible to store whole tuples, or just their TIDs, which minimizes storage cost for the hash table.

Depending on the hash function, it is possible that multiple different values are mapped to the same hash value. This property is called *collision* and influences the performance of the hash join, as we describe later on.

In the second phase (the *probe phase*), the second relation is sequentially scanned and each tuple is hashed with the same hash function the hash table was created with. Then, the resulting hash value is used to retrieve possibly matching tuples from the hash bucket with the same hash value. After that, the real key values of each retrieved candidate from the hash table have to be checked for a match to the current tuple of the second relation, because of the possibility of collisions.

```
1  Input: Relation A, JoinAttribute jA, Relation B, JoinAttribute jB
2  Result: Relation C
3  Build phase:
4  foreach Tuple a in A do
5  |  HashTable.insert(H(a.jA),a);
6  end
7  Probe phase:
8  foreach Tuple b in B do
9  |  // Retrieve possible matches for a from the hash table
10 |  foreach Tuple a in HashTable.get(H(b.jB)) do
11 |  |  if a.jA = b.jB then
12 |  |  |  // Keys match → join tuples
13 |  |  |  c:= join(a,b);
14 |  |  |  C.insert(c);
15 |  |  end
16 |  end
17 end
18 return C;
```

Algorithm 2.3: Hash-join algorithm – adapted from [ME92].

In Algorithm 2.3, we formulated algorithms for both phases of the hash join. The computational complexity of hash joins is $\mathcal{O}(n+m)$, because every relation is scanned once. A significant impact on the performance has the used hash function. If the hash function causes numerous collisions, many tuples have to be compared in the probe phase which decreases performance. Furthermore, it is important to choose the right relation for the build phase. Optimally, the smallest relation should be taken for building the hash table [ME92], because it reduces storage costs.

2.3.5 Sort-Merge Join

Another possible strategy for joining two relations is the sort-merge join. Like the hash join, the sort-merge join consists of two phases. In the first phase, both relations have

```
1  Input: Relation A, JoinAttribute jA, Relation B, JoinAttribute jB
2  Result: Relation C
3  Sort phase:
4  sort(A,jA);
5  sort(B,jB);
6  Merge phase:
7  // Initialize first tuple of b.
8  b:= B.begin();
9  foreach Tuple a in A do
10     while a.jA > b.jB do
11         b:= B.next();
12     end
13     if a.jA = b.jB then
14         // Keys match → join tuples
15         c:= join(a,b);
16         C.insert(c);
17     end
18 end
19 return C;
```

Algorithm 2.4: Sort-merge-join algorithm – adapted from [ME92].

to be sorted, if they are not already in a sorted order. The second phase is the merge phase, where the sorted relations are sequentially scanned to retrieve matching tuples.

In Algorithm 2.4, we present a simplified version of the sort-merge join. For the merge phase of the simplified algorithm, we sequentially compare the values of the join attribute. If the values are the same, we have to merge these tuples. Otherwise, we have to read the next tuple of the relation with the smaller value. This implementation assumes, that there are no duplicate values in the join attribute. Otherwise, we would have to add a nested-loops join for joining tuples from `A` and `B` with duplicate key values.

The performance of the sort-merge join highly depends on the sorting step, which in general has a complexity of $\mathcal{O}(n * log(n))$ [ME92]. If one of the relations, or both, are already sorted or indexes have been constructed, which can retrieve tuples in a sorted order on the join attribute, only the complexity of the merge step has to be considered, which is $\mathcal{O}(n + m)$.

In this chapter, we gave an extensive overview of the traditional database design and components that are involved in the processing of database queries. A severe impact on the query performance has the access gap between primary and secondary memory and has to be considered when executing operations. A famous operation in databases is the natural join. Hence, we presented suitable strategies to perform a natural join, namely nested-loops and block-nested-loops join as well as hash join and sort-merge join.

In the remainder of this thesis, we show the influence of new hardware on the database architecture, processing steps inside a database and finally, its influence on the performance of our presented join strategies.

3. Recent Advances in Database Technology

In the last 10 years, there have been steady improvements and novelties in hardware. This new technology influences the way database systems work and may favor different operations or algorithms. For a detailed consideration of the impact of new hardware on databases, we present recent advances in technology in the following chapter.

We start with recent advances in in-memory databases and in Section 3.2, we show the technology of multi-core CPUs and co-processing devices, such as GPUs and FPGAs. Finally, we present SSDs bridging the access gap and consequently representing a possible alternative to in-memory databases.

3.1 In-Memory Databases

Enormous improvements in hardware over the last decades have raised the importance of in-memory databases. However, these improvements are not as high in access time as in memory capacity, since there is a trade-off between capacity and access time in RAM [BMK99]. As a result, nowadays for many applications it is a reasonable approach to have all data to be processed in RAM, which changes the memory hierarchy.

When using in-memory databases, the access gap between HDD and RAM can be disregarded, since all data is in RAM. However, fully exploiting the computational potential of the CPU is not achievable, since CPU processing speed is much higher than access latencies of cache and RAM. The gap between processing speed and access latency, which is also called the *memory wall*, is going to grow [BMK99]. As a consequence of the memory wall, Boncz et al. improve the processing model of databases to match cache requirements in *MonetDB* [BMK99].

3.1.1 Column-Oriented Storage for In-Memory Databases

A common approach to improve processing performance in an in-memory database is to change the storage model from row-oriented storage to column-oriented storage [BKM08]. There are several advantages when using column-oriented storage schemes (*column store*).

- On the one hand, good compression rates exist for column stores. Data values of one column are very similar to each other whereas data values in one row are often varying strongly, especially in type and domain. Thus, it is more effective to compress data of one column than data of one row [RDFH12]. Numerous compression techniques are presented in [SSH11].
- On the other hand, query execution may benefit from a column-oriented storage. Considering queries with projections, unnecessary columns can be filtered out in early phases. Furthermore, I/O-costs are reduced and cache overflow is prevented, because processed data items are much smaller [BKM08]. To illustrate these benefits , we depict the column-oriented execution of the projection and selection on the `customer` table of Figure 2.2 in Figure 3.1.

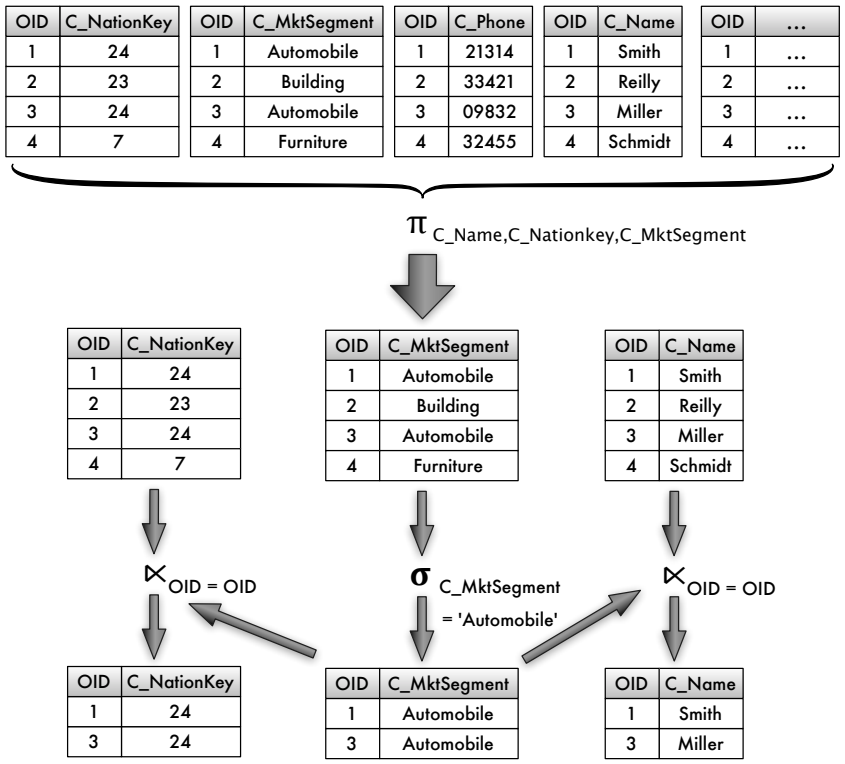


Figure 3.1: Projection and selection in a column-oriented database system.

Storing tables column-oriented instead of row-oriented, however, is not only beneficial. For example, when the whole table has to be retrieved, tuple reconstruction is a task with high effort in column stores compared to row stores [BKM08, KSS12].

As a consequence of the changed storage model for in-memory databases, the processing model is adapted as well. The impact of this new processing model is sketched in the following.

3.1.2 Operator-at-a-Time Processing Model

The traditional way of executing operations in a DBMS is *tuple-at-a-time*, which was extensively covered in the volcano model proposed by Graefe [Gra94]. Here, one tuple passes several operators in a pipelined-way until it is added to the result table [BKM08] and after that, the next tuple passes the pipeline. As a consequence, one processor may have to cache all operations of the pipeline which may lead to an overflow in the instruction cache.

Nevertheless, in the traditional database architecture and memory hierarchy, overflowing caches represent no performance issue, since the bottle neck is located between HDD and RAM. As this bottle neck shifts to cache usage considering in-memory databases, an improved processing model is proposed, which is called *operator-at-a-time*.

In the operator-at-a-time processing model, all data which has to be consumed by one operator is processed completely and then passed to the next operator. For operator-at-a-time, processed data has to be as small as possible to avoid cache overflow, consequently column stores are the preferred data representation. The change of the processing model influences algorithm design as well. Shatdal et al. propose five optimizations of algorithms to make them more cache conscious [SKN94]. Important points are reducing the data which has to be accessed for example by projections, blocking or partitioning of data so that it fits in cache, or combining two or more loops that run over the same data separately.

Especially join algorithms benefit from a cache-conscious algorithm design. Shatdal et al. acknowledge about 10% smaller execution time for cache-conscious join algorithms [SKN94]. Thus, an impact of storage location on the join performance and algorithm design has been shown.

3.2 Multi-Core CPUs and Co-Processing

Algorithm design is not only influenced by the storage architecture and model, but even more severely by the processing device, because they dictate how data is processed and which operations are efficiently supported.

There are several possible processing devices that have to be considered, namely *multi-core CPUs* as primary processing devices, and *GPUs* and *FPGAs* as co-processing devices. At first, we review processing capabilities of multi-core CPUs and continue with GPUs and FPGAs.

3.2.1 Parallelization in Multi-Core CPUs

The processing capabilities of modern CPUs are enormously increasing. To reach high performance, CPUs offer a large degree of parallelism by having multiple cores on a single chip. Furthermore, they support multiple hardware threads on each core and offer *single instruction multiple data (SIMD) computation*. SIMD means that one instruction can be applied to multiple vectors of currently 128-bit in parallel [KSC+09]. Thus, an operator consumes several data items in one run instead of iterating over data one by one.

To exploit the processing capabilities of multi-core CPUs, *thread-level* and *data-level parallelism* have to be considered when engineering algorithms.

- For thread-level parallelism, database operations should be able to be executed in parallel on multiple threads. Hence, data has to be able to be partitioned into corresponding threads without incurring concurrency on data, since this would lead to synchronization issues [KSC+09].
- Exploiting data-level parallelism includes to fully load SIMD processors. For processing chunks of data in parallel by one operator, the data has to be contiguously stored in cache or RAM. Otherwise, there is a high overhead, because of the memory wall [KSC+09]. Consequently, storing data column-wise helps increasing data-level parallelism and is a good choice for multi-core CPUs.

Considering the performance of join algorithms, Kim et al. evaluate hash join and sort-merge join as the most promising join algorithms on multi-core CPUs [KSC+09]. Their evaluation shows, that hash-join performance is superior to the one of sort-merge join. However, sort-merge-join performance is increasing when the size of tuples decreases, since more tuples can be kept in the cache for comparisons. With this in mind, they assume that the sort-merge join could have a better performance than the hash join for small sized tuples using 256-bit SIMD and could outperform the hash join with 512-bit SIMD.

Since high-performing SIMD seems to be a key capability for good parallelism as well as for a change in the performance hierarchy of join algorithms, we review a device offering this capability, namely the GPU.

3.2.2 Co-Processing of Database Operations on GPU

In the last decade, query processing on *graphics processing units (GPU)* as co-processing devices attracted much attention in literature [GLW+04, FHL+07, HYF+08, KLMV12]. The availability of *general-purpose graphics processing units (GPGPU)* and their increasing processing power have emerged their usage also in database applications.

In fact, GPGPUs tend to offer in average the same quantities in throughput as modern CPUs. This makes GPUs very attractive as a processing device [LKC+10]. To achieve

good processing power, a GPU provides parallel lower-clocked execution capabilities on several hundreds of *single instruction multiple data (SIMD) processors*. In contrast, CPUs allow an out-of-order execution of branches in a program while using much less cores. Thus, GPUs are not suited for branching, because some GPU cores will stay idle while the others execute the branch [BBR+12].

In general, the processing on the GPU consists of three steps. At first, the host code allocates enough memory in the GPU RAM so that input and output data fit and copies the input data from RAM to occupied GPU RAM. In the second step, the host code is started on the GPU, consumes input data and stores the result in allocated GPU RAM. As third step, data is copied back to main memory [HYF+08]. As a result of this procedure a copy overhead is introduced. Consequently, execution decisions have to be well considered and first approaches for load balancing of database operations between GPU and CPU are proposed [LKC+10, BBR+12].

Considering performance of different join strategies, He et al. make a comparison between indexed nested-loops join, non-indexed nested-loops join, hash join and sort-merge join [HYF+08]. They show that performance improvements of factor two to seven can be achieved in join processing on GPUs compared to an execution on multi-core CPUs.

The best performing join in the evaluation of He et al. is the indexed nested-loops join. Since we consider only ad-hoc join algorithms, this strategy is out of scope. The second best performing join is the hash join, closely followed by the sort-merge join. Far behind them is the non-indexed nested-loops join. However, a comparison of the impact of parameters and workload is not given.

Considering the assumptions of Kim et al. about CPUs with wider SIMDs, a GPU should favor the sort-merge join as well if their vector register width extends to 256- or 512-bit. Currently, GPUs support 128-bit vectors, so further evaluations have to be done in future.

3.2.3 FPGA as Co-Processing Device

A promising new processing device is the *field-programmable gate array (FPGA)*. An FPGA can be seen as a number of logic gates whose wires can be programmed by using software making it usable as a hardware-accelerated implementation for computation or control tasks [MT09].

The processing in an FPGA takes place in several *lookup tables (LUT)* having k inputs (k usually between 4 and 6) and one output. Each LUT is programmable to implement any Boolean function with k inputs. With the help of a programmable *interconnect fabric*, the behavior defined in software is mapped to the functionality of LUTs as hardware.

FPGAs offer a high parallelism and can be used for processing data streams. Especially in databases, selections with many Boolean operators can be processed together in one

clock cycle of the FPGA instead using several cycles on the CPU [MT09]. However, FPGAs work on lower clock speeds and processing capability is also limited by the number of available lookup tables.

Considering the execution of join algorithms on FPGAs, recent research of Vaidya and Lee propose a join implementation for coarse-grained FPGAs looking similar to a block-nested-loops join [VL11]. Several processing units (between 16 and 256) in the FPGA are populated with data of one relation and step by step, tuples of the second relation are rotated through them to find join partners and output them.

Their evaluation shows speed ups of 6 to 100% on FPGAs in contrast to CPUs depending on the number of the processing units in the FPGA. Furthermore, when the amount of data increases, FPGA speed ups increase as well, because the time for programming the FPGA amortizes with increasing table sizes. As a consequence, joining small tables might not be efficient on FPGAs [VL11].

Considering their used join algorithms, they did only consider one join strategy and maybe there are better strategies possible for an FPGA. Hence, a performance evaluation of different join algorithms has to be done and stays open for future work in the area of co-processing.

3.3 SSDs to Bridge the Access Gap

In Section 2.2, we already introduced the access gap, which means that access time between primary and secondary memory differs in an order of 10^5. The access gap troubles the processing power, since data is faster processed than new data is retrieved from the secondary memory. The reason for the access gap is the inferiority of hard disk drives in access times compared to the RAM. Thus, for scenarios, where more data is processed than available RAM, we need an alternative to an in-memory system. To bridge the access gap, *solid-state disks (SSD)* may be an opportunity to accelerate accesses.

3.3.1 Properties of SSDs

In contrast to HDDs, an SSD contains no moving parts. SSDs consist of *NAND-flash memory chips* and controllers that provide a block-wise access to storage cells of the chip. The memory is divided into blocks which are subdivided into a number of flash pages. Each storage cell of a page is initially set to 1. To store a value, the SSD switches necessary bits to the value 0. If the value of a written flash page has to be changed, all pages of a whole block have to be erased and their content set to 1. After that, a page of the erased block can be rewritten by setting necessary memory cells to 0 [DP09].

Time critical operations on an SSD are not reading and writing of pages, but erasing of blocks. According to [APW+08] depending on the used device, reading a page takes approximately $125\mu s$ in sum and writing a pre-erased page requires $300\mu s$ for the whole execution. In contrast to that, erasing a block of pages to allow further rewriting is substantially more expensive taking $1.5ms$ [APW+08].

Another limitation of SSDs is the number of possible erase cycles, because constant writes on the same NAND flash eventually wears it out which means that a complete reset of a block is not possible anymore. In numbers, many devices allow up to 100,000 write cycles per memory cell. Consequently, writes have to be distributed over all available blocks by the SSD to minimize an unbalanced aging of single memory cells of the device. For this, *wear-leveling* is implemented inside the SSD controller to ensure an even aging of memory blocks. With wear-leveling, the SSD has a method to choose which block to erase depending on how often the block was rewritten before [APW+08].

Despite given disadvantages of SSDs, the read and write performance of an SSD is highly superior to those of HDDs. Especially random reads are much faster on an SSD than on the HDD, because there are no moving parts involved when reading blocks or pages from the SSD. Even limited lifetimes of SSDs do not pose a severe disadvantage, since overwriting a 32GB flash disk with 30MB/s a $100,000$ times would take $3^1/_2$ years. Thus, we discuss the usage of SDDs as part of the memory hierarchy in the next section.

3.3.2 Adding SSDs to the Memory Hierarchy

The access gap exists between the RAM with access times less than 100 nanoseconds and HDDs operating in orders of several milliseconds. With this in mind, SSDs are valuable devices to fill this gap, because they offer access times of hundreds of microseconds which is in the middle of access times of RAM and HDD.

In general, there are two possibilities for using SSDs in the memory hierarchy. On the one hand, SSDs could be used as an extension of the secondary storage *(extended disk)* and on the other hand, SSDs could represent an own component in the primary memory *(extended buffer pool)* [Gra07]. Both variants will be reasoned in the following.

SSD as Extended Disk

A considerable concept using SSDs would be to support HDDs in persistently storing data. This increases the overall performance, since data that is accessed many times (*hot data*) is stored on the SSD while data which is infrequently used (*cold data*) resides on the HDD. The effort for the database system is to manage the storage location of data and if access patterns change, cold data has to be transferred to the HDD and hot data loaded to SSD.

Especially the impact of SSDs on join algorithms is an interesting topic, since using SSDs their performance have a greater tendency to become CPU-bound (rather than I/O-bound) [DP09]. Furthermore, Do and Patel propose that algorithms should favor sequential writes instead of random writes when using SSDs, since random writes cause more blocks to be deleted [DP09]. Consequently, an algorithm should write its data sequentially down even if this incurs a random access when reading it. Another interesting observation of them is that the execution time of random writes varies highly, thus they favor sequential writes.

Graefe is particularly optimistic that extended disk is the best configuration for databases, since all data is always persistently stored [Gra07] and uncommitted data is not stored

on persistent storage. Especially for checkpointing, which means a consistent stage of the database has been made for backup issues; this configuration is the preferred one. Although SSDs as extended buffer pool seem to have disadvantages in checkpoint scenarios, we argue their advantages in query performance in the following.

SSD as Extended Buffer Pool

Instead of using SSDs as an additional persistent storage device, their fast access time makes them a considerable device for cached data. To this end, data that was replaced in RAM, because of its little use, is at first stored on SSD. Just when it has to be replaced on SDD, it is stored on HDD.

As a consequence of the usage as buffer pool, there have to be data structures in-memory that manage the residence of data on SSD or their replacement to disk. For this, Graefe proposes an LRU priority queue which orders the data by their last accesses [Gra07]. The least-recently-used approach is a reasonable replacement strategy, since properties, such as the number of accesses, do not change once the data is on the SSD. Furthermore, a hash table for finding the storage location of data on SSD has to be held in RAM.

Considering the usage of SSDs as an extension of the buffer pool in a database implicates some drawbacks. As we have mentioned before, database systems create checkpoints of current data to backup the database in case of upcoming failures. When a checkpoint is to be created, changed data in the buffer pool is written to disk, to have a consistent image of the database. Thus, when checkpoints are very frequently done, cached data on SSDs is forced to the disk very often. Consequently, caching benefits decrease, since the buffer pool is renewed after every checkpoint.

Nevertheless, we argue that especially in analytical queries, for instance in data warehouses (so-called *OLAP transactions* [KSS12]) which may have long execution times and include intensive processing, SSDs extending the buffer pool are a valuable processing support. Especially join algorithms may benefit from the usage of SSDs as extended buffer pool.

3.4 Conclusion

After representing several advances in modern hardware, we want to summarize the assumptions that were given in research on the join performance on modern hardware. With this, we fulfill the first contribution of our work. We found the following assumptions for different storage and processing devices:

- Storing data in memory favors a column-wise storage of data and improves the operator-at-a-time processing model for join algorithms. However, there is no evidential influence on the join strategy when using in-memory databases. Solely the algorithm design has to be adapted to fulfill cache-consciousness, because of the memory wall.

- Multi-core CPUs offer high capabilities for data-level and thread-level parallelism which have to be exploited extensively in the algorithm design. Considering the join strategies, current multi-core CPUs favor hash joins. However, with increasing SIMD capabilities of CPUs, sort-merge joins should become superior.
- The parallelization of modern GPUs with highly parallel SIMD but restricted branching capability favors hash-join as well. However, if SIMD capabilities expand similar to those of CPUs, it is likely that sort-merge join becomes superior to hash joins.
- FPGAs are an upcoming alternative to accelerate database operations using new hardware. FPGAs could be programmed to execute block-nested-loops join more efficiently. However, there is a need to evaluate the performance of different join strategies on FPGAs.
- SSDs could be used in databases to bridge the access gap between RAM and HDD. However, comprehensive performance evaluations of join algorithms using SSDs as extension of the buffer pool are needed.

To conclude, there are many open questions concerning the performance of different join strategies on new hardware. In this work, we are evaluating the join performance for ad-hoc joins on different storage devices, since there are still open questions, especially when using SSDs. To have a conclusive evaluation, we have to identify possible impact factors on join performance that are covered in the following chapter.

4. Preliminaries for Database Operation Evaluation

Performance evaluation of algorithms is a difficult task, because there are numerous factors, which influence performance. Considering database operations, there is not only an impact of hardware, but also of functionality that has to be provided by the database system. For example, buffer management influences the performance of operations, since it manages how data is accessible for the operation. To have an overview of possible impact factors on database operations, we present them in the following section. This will help us on the one hand, to identify variable components in our following implementation and on the other hand to provide a number of variants that could be evaluated. Consequently, our first contribution of this chapter is to show a comprehensive and well-structured overview of impact factors on database operations.

Although a comprehensive evaluation of database operation performance requires the consideration of all impact factors, not all of them are evaluable in their entirety, because some of them (e.g., the buffer pool size) imply an unlimited amount of possible values. As a consequence, we have to limit our considerations to a few selected factors. Hence, our second contribution is a reasonable discretization of the impact factors to enable a first evaluation of the performance of database operations.

4.1 Impact Factors on Database Operations

In this section, we describe impact factors on database operations that have to be considered for a performance evaluation. Since a complete model of all impact factors is hardly achievable, we argue that our model covers at least the most important areas to be considered. In fact, our extensive literature review assures comprehensiveness of high levels in our categorization. To this end, our model includes traditional design factors described in Chapter 2 as well as factors of new approaches which are derived from the survey in Chapter 3.

Considering impact factors on database operations, there are different combinations between them. For instance, some factors are obligatory while others are optional. To present these impact factors in a well-structured model and their relation in an easily understandable way, we rely on *feature models*. Feature models are the state-of-the-art to depict variability in the context of *software product lines* [BSRC10] and, thus, a good solution for describing our impact factors on database operation performance. A feature model is a set of features which are arranged in a hierarchical way and they offer several possibilities to model relationships between features. We introduce necessary notations in the following and refer to the work of Batory for a comprehensive overview of possible relations [Bat05].

The advantage of using feature models as representation is that we can compute a number of possible configurations of our evaluation. In Figure 4.1, we depict a coarse-grained variant space using a feature diagram for visualizing feature models. In this representation, a rectangle denotes a feature of the evaluation, which means it is an impact factor that may be taken into account when doing the evaluation. All of the four child features are mandatory which means if the parent is chosen (the root in this case), they are chosen as well.

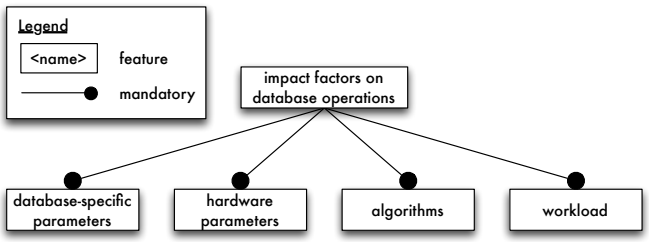


Figure 4.1: Groups of impact factors.

For a first classification of the impact factors, we define four groups of impact factors. Arising from our overview in Chapter 2 , we identify *database-specific parameters* and *algorithms* as impacting components. Furthermore, new challenging *hardware* and its strengths and weaknesses are presented in the previous chapter and have to be included in our consideration as well. An impact factor that strongly influences the algorithm choice is the *workload*, because some algorithms may be in favor considering specific properties of data. With the given coarse-grained categorization, we argue that all factors considered in the literature fall into one of these categories.

Since the four impact factors are always present when executing evaluations of database operations, we model them as *mandatory features*. As a result, the number of possible configurations does not change considering these four features, because they do not provide any variability. However, they are important, because they serve as a categorization of the underlying features, which are presented in the following.

4.1.1 Database-Specific Factors

As database-specific factors, we define impacts that are caused by the database management system. These impacts include functionality that has to be provided to fulfill the specification of a database system, which were also introduced in Chapter 2 and Chapter 3. From these chapters, we extract typical impact factors of database systems and depict them in the feature diagram in Figure 4.2.

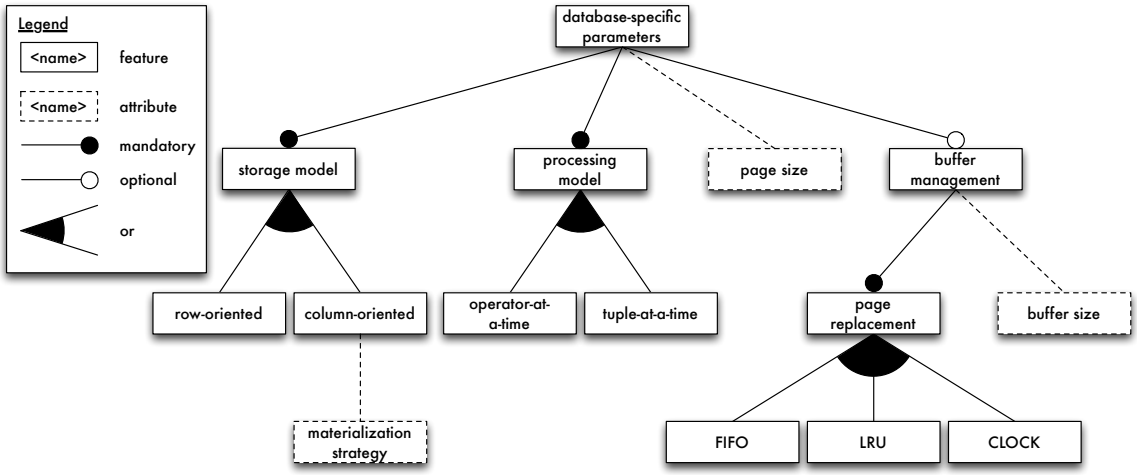


Figure 4.2: Factors implied by chosen database system design.

Two typical design choices, which impact the functionality of a database system, are the storage model and the processing model. Hence, they are mandatory features in the diagram to express their necessity. We present their characteristics in the following.

Storage Model

For the storage model, we decided that it is possible to store data in a row- or column-oriented way. Furthermore, a combination of both storage models is possible, as the work of Lübcke et al. indicates [LSS13]. They argue that it is possible to store data redundantly in a hybrid storage system of row- and column-oriented tables. Consequently, our variant space includes the three possibilities of single row- or column-oriented storage as well as a hybrid storage system, which is denoted by an *or relationship*. Hence, we have to be able to evaluate our database operations under different storage models.

Considering the column-oriented implementation, an open issue is at which point in the query execution the database management system reconstructs the tuples. Abadi et al. present early and late materialization for points of reconstruction [AMDM07]. The differences between these two strategies are the intermediate results. In an early materialization, values of a tuple of a processed column are added to an intermediate tuple. Thus, after the last operation, the intermediate result is the final result. In contrast, late materialization uses compressed data structures or row IDs to reference tuples, which provides a more effective cache utilization, since intermediate results occupy less

storage space. However, to retrieve the result, values of tuples have to be retrieved and a second access to data is incurred.

Abadi et al. present different types of early and late materialization depending on the executed operation [AMDM07] and, consequently, we cannot model it as a Boolean feature. Since a materialization can take place at any point in the query plan, the materialization strategy does not represent a feature which can either be taken into account or not. Hence, we use the concept of *extended feature models* and model the materialization strategy as an attribute to indicate that there are several forms [BSRC10].

Processing Model

Regarding hybrid storage systems, a hybrid processing model is also considerable. A join may either compare and join single tuples, whole columns, or a whole column with single tuples. As a consequence, tuple-at-a-time [Gra94] and operator-at-a-time [BMK99] operations are in an *or relation* as well. This impacts our considerations significantly, because we have to consider access to single tuples as well as to a whole block of tuples or columns.

Page Size

Another important property of databases is the page size. Data of a database system, such as columns or rows of tables as well as index information, is stored on pages that are synchronized with the persistent storage. Since pages are the container of the storage system, they specify the granularity of access. Small pages may incur many reads, but offer a more fine-granular and selective access than bigger pages. Consequently, the amount of storage space per page is an important impact factor on the performance of database operations. Since the page size can take an arbitrary numerical value and does not represent a Boolean feature, we model the page size as an attribute to indicate that it may take several values.

Buffer Management

A further database-specific impact factor is the buffer management. The buffer management is an optional feature in our feature model, because there are configurations that do not require a buffer manager. Especially an in-memory database operates without a buffer manager, since all data is already in RAM. Every buffer manager has a fixed size of its buffer pool, generally defined as the number of fitting pages. This property is also represented as an attribute, because buffer sizes may vary from system to system. Another sub-feature of the buffer management is the page replacement strategy, since the buffer manager has to identify pages that have to be replaced if the buffer size is exceeded and an additional page has to be loaded. As page replacement strategies we choose LRU, FIFO, and CLOCK in our feature model. Of course, there are further page replacement strategies introduced in literature and we refer to Saake et al. for a comprehensive overview [SSH11]. Nevertheless, we list only these three strategies as representatives here, since they are easy to implement and leave further extensions open

for future work. Furthermore, we argue the relation between them is an *or relation*, because there may be several buffers. More specifically, we can choose a different page replacement strategy in the SSD buffer than the one chosen for our main memory.

4.1.2 Hardware Parameters

Another group of impact factors that influences the performance of database operations are hardware parameters. Especially changing the hardware may influence processing capabilities as well as algorithm design as we have already emphasized in Chapter 3. As a result, we depict the two groups of hardware specific factors in Figure 4.3, namely the used processing devices as well as storage devices.

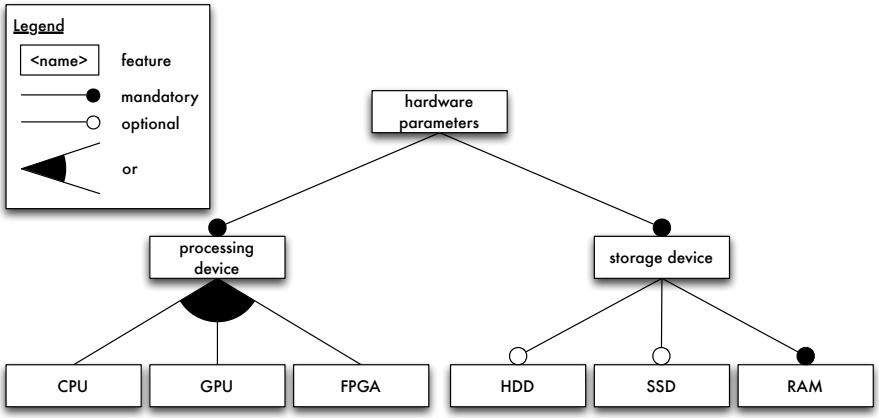


Figure 4.3: Hardware-specific impact factors.

Processing Device

As processing devices, we categorize hardware components that process data in order to execute a database operation. For this, we list CPU, GPU, and FPGA. We argue, that it is possible to choose multiple devices for processing, because, for instance, in the sort-merge join, we could execute the sorting on the GPU while the CPU is responsible for comparison and joining of tuples. Thus, we model the processing devices CPU, GPU, and FPGA with an *or relationship* and this way, we support modern co-processing approaches [MT09, BBR+13].

An aspect that has to be considered for future work is the capability for parallelization, because different processing devices provide different types of parallel execution. Considering CPUs and GPUs, the CPU offers branching capability in execution while the GPU executes one operation on all cores in parallel. Furthermore, whether the code is written for single-threaded or multi-threaded execution is a point to be addressed in future work. Thus, our work presents an initial step for an evaluation of database operations on varying hardware.

Storage Device

Additionally, storage devices influence the performance of database operations, because they hold data to be processed. We consider HDD, SSD, and RAM as storage devices. The traditional way consists of HDD as persistent storage and RAM as storage device to provide access for the CPU, but also a combination of all three devices is possible. Nevertheless, there is no storage system working without RAM and, thus, it is a mandatory feature. In contrast, storing data on HDD or SSD is optional and included in our model as an optional feature.

When data is stored on HDD or SSD, the scenario implies that the RAM is limited and does not provide enough capacity for all our processed data. Consequently, a buffer manager has to manage the available space in RAM. This necessity for a buffer manager will be expressed by a *cross-tree constraint* in the whole feature diagram.

An important feature selection for our work includes all three available storage devices. In this case, a limited part of the data that is stored on HDD fits into RAM. While the execution of an operation, the buffer manager has to replace unused pages to continue processing of new data. For this, the pages are written to the SSD acting as an additional buffer pool including an own buffer manager. This may be advantageous for those systems, where data exceeds the amount of available SSD storage and, thus, the SSD can only be used as a buffer providing faster access to often used pages than the HDD.

Limitations of our Model

Considering the storage and processing devices, there are still properties that have not been considered in our feature model. Each storage device has a capacity and latencies as well as a limited bandwidth. Nevertheless, we excluded them from the feature model, because depending on the use case, we do consider different properties. For example, the capacity of an HDD does not matter for the evaluation in a traditional system, because it has to provide only enough space for storing the tables. In contrast, the storage capacity of RAM is important, when just a part of the data fits into it. Furthermore, considering in-memory databases, the RAM capacity does not have to be considered in the performance evaluation of database algorithms, because of the assumption that all data fits into RAM.

Furthermore, techniques, such as *RAID* are taken out of consideration. A RAID system balances access time using a cluster of storage devices with redundant data. Since the behavior of RAID influences access times, these systems are future work for performance evaluation of changing hardware.

4.1.3 Database Operations and Their Algorithms

Not only hardware and database parameters influence the performance of database operations, but also more considerably, the chosen algorithm influences the performance. Boncz et al. identify *selection*, *join*, *sort*, and *aggregate operations* as important database operations [BMK99] and, thus, we consider these four operations in Figure 4.4.

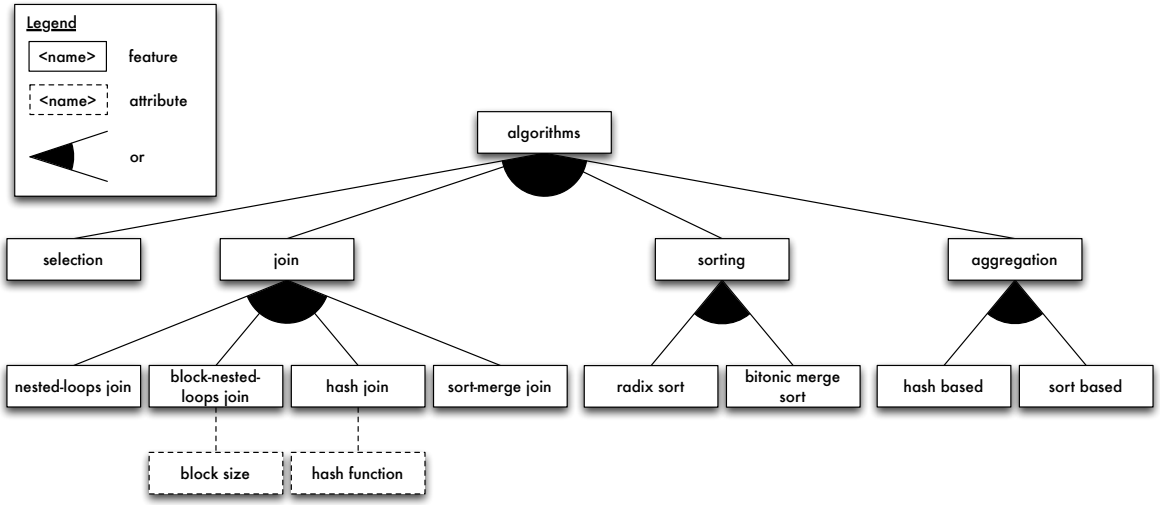


Figure 4.4: Parameters of algorithms.

Considering the selection operation, suitable data structures have to be provided. Since we do not cover index structures in this work, we exclude the selection operation from further evaluations and leave it open for future work. For join operations, we consider nested-loops, block-nested-loops, hash, and sort-merge join, which can be evaluated to find the best strategy for a join. There are further properties that influence the performance of block-nested-loops join or hash join, which is the block size for the former and the hash function for the latter. In general, the sort-merge join requires a good implementation for sorting, because sorting produces the computational effort with a computational complexity of $\mathcal{O}(n * log(n))$. However, the sort algorithm is another feature in our feature model and, thus, a connection between them would be necessary which could be expressed in a multi-product-line language, such as *VELVET* [STSS13]. Thus, for future work, it could be helpful to model our variant space as a multi product line.

Based on our literature review, we propose radix sort [BMK99] and bitonic merge sort [FHL+07] as important sorting strategies and include them in our feature diagram for possible evaluations. Furthermore, aggregations can be implemented as hash-based or sort-based algorithms [BMK99] while the sort-based approach depends again on the best sorting algorithm.

4.1.4 Impact Factors of Workload

An important group of parameters represents the workload, because some workloads may favor different strategies of an operation. For example, Kim et al. state that skewed data troubles the performance of the hash join, because the data does not distribute equally over all hash buckets [KSC+09]. Hence, the data distribution is an important property of the workload.

In addition, table sizes impact the performance of database operations. For example, considering hash join, if one table has significantly less tuples than the other, the smaller table has to be taken for building the hash table [ME92]. Thus, table sizes have to be included in our feature model.

The last two features of the workload regard the property how data has been pre-processed. On the one hand, data can be stored in a sorted way. With this in mind, the sorting phase of sort-merge join can be skipped which gives this strategy an important advantage. On the other hand, an index on the data may favor different algorithms. For example, He et al. identified the nested-loops join as the best join strategy on the GPU for indexed data [HYF+08].

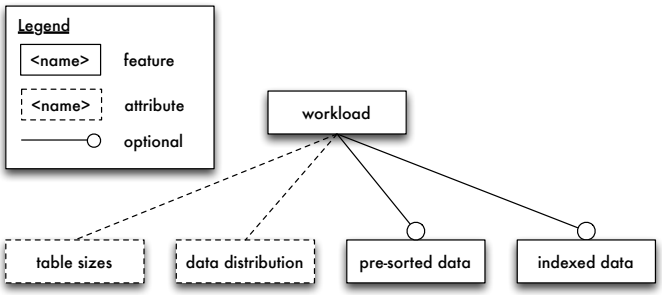


Figure 4.5: Impact factors of specific workload.

In Figure 4.5, we show the feature diagram for the workload. We represent table sizes and data distribution as attributes, because these properties have an unlimited amount of possible values. In contrast, data that is pre-sorted or indexed can be represented as optional features, because it can be determined whether the property is fulfilled.

4.1.5 Summary of Variant Space

For the complete feature diagram of impact factors, we merge the feature diagrams, presented before, in Figure 4.6. Furthermore, we add a constraint to exclude some of the variants that are not possible. The constraint is $SSD \vee HDD \Leftrightarrow buffer\ management$, which expresses that if we store our data on an SSD or HDD, we also have to use the RAM with a buffer manager to make data available for processing.

One result of building a feature model is that the number of possible configurations is computable. However, not all of our considerations can be taken into account. The aspect of using attributes in the feature model disables the possibility to count the number of possible variants, because an attribute often represents an unlimited range of values leading to an infinite number of variants. Hence, for computing the number of possible variants, we rely on Boolean features only. Consequently, our computed number of variants differs from the actual one that includes attributes.

For modeling the feature model, we use FeatureIDE [TKB+13] for Eclipse and export the feature model to use it in S.P.L.O.T.[1] [MBC09], an online automated analysis tool

[1] http://www.splot-research.org

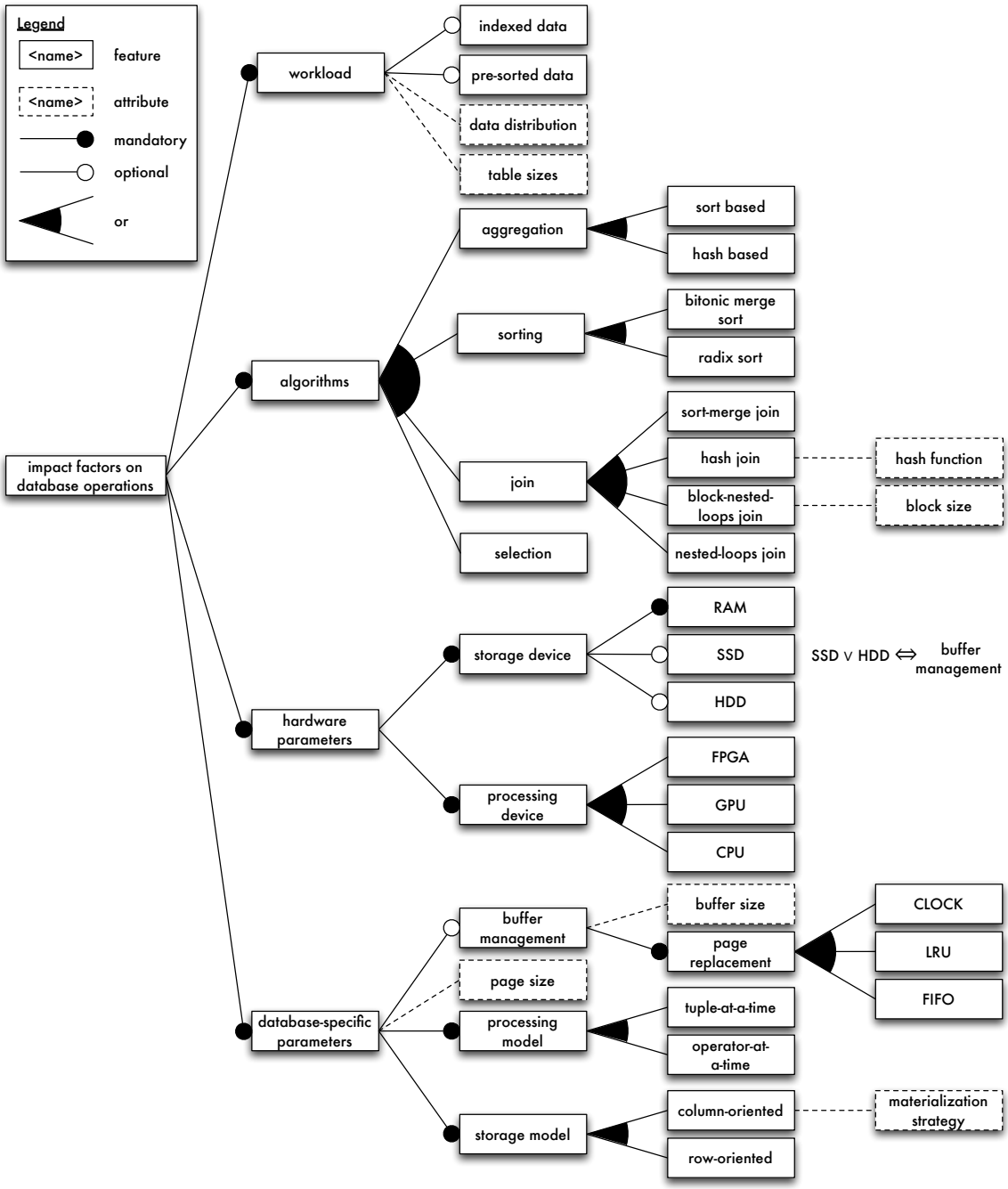


Figure 4.6: Summary of impact factors on performance of database operations.

for feature models. S.P.L.O.T. computed 3,734,388 possible variants. Since we are not able to evaluate such a high number of possible variants, we have to limit our considerations to a selected number of features. Thus, we present a reduced feature model in the following section.

4.2 Discretization of Variant Space

Our feature model of the last section shows a big amount of impact factors on database operation performance. On the one hand, a broad identification of impact factors is of high importance, because our evaluation model should offer the possibility to extend it to support every variant. On the other hand, in the current state, we have to limit our model, because an evaluation of the whole variant space within this work is not feasible.

For a reduction of the space of possible variants, we limit our consideration to selected features and present these limitations in the following. The resulting feature model is depicted in Figure 4.7. Features that are not considered are shown in gray color and only features to be evaluated are still visible with black borders and labels.

Database-Specific Parameters

For our evaluation, we limit the database-specific parameters to those of the traditional database systems, because its functionality is well known and effects on performance are well explainable. Hence, we evaluate algorithms on a row-oriented storage system and process one tuple after another instead of operator-at-a-time processing. As a page replacement strategy, we reduce our consideration to the CLOCK algorithm, because LRU and FIFO are said to not scale for database applications [SSH11]. However, impact factors, such as buffer size and page size are still taken into account in our evaluation and we choose different reasonable and often used values for them.

Hardware Parameters

The used hardware in our evaluation is, at the moment, limited to one given system. The processing is executed on the CPU and as storage system we have the choice to use either HDD, SSD, RAM, or their possible combinations. Finding suitable algorithms for different processing devices is open research for future work.

Apart from the extended disk approach proposed by Graefe, we focus on the idea of an extended buffer pool when we use the SSD in combination with an HDD and RAM. The extended disk approach is partly evaluated when using the SSD only, under the condition that hot data to be joined is stored on SSD.

Algorithms

In our work, we limit the database operations to the most important one – the relational join [BMK99]. This limitation decreases our variant space significantly and breaks it

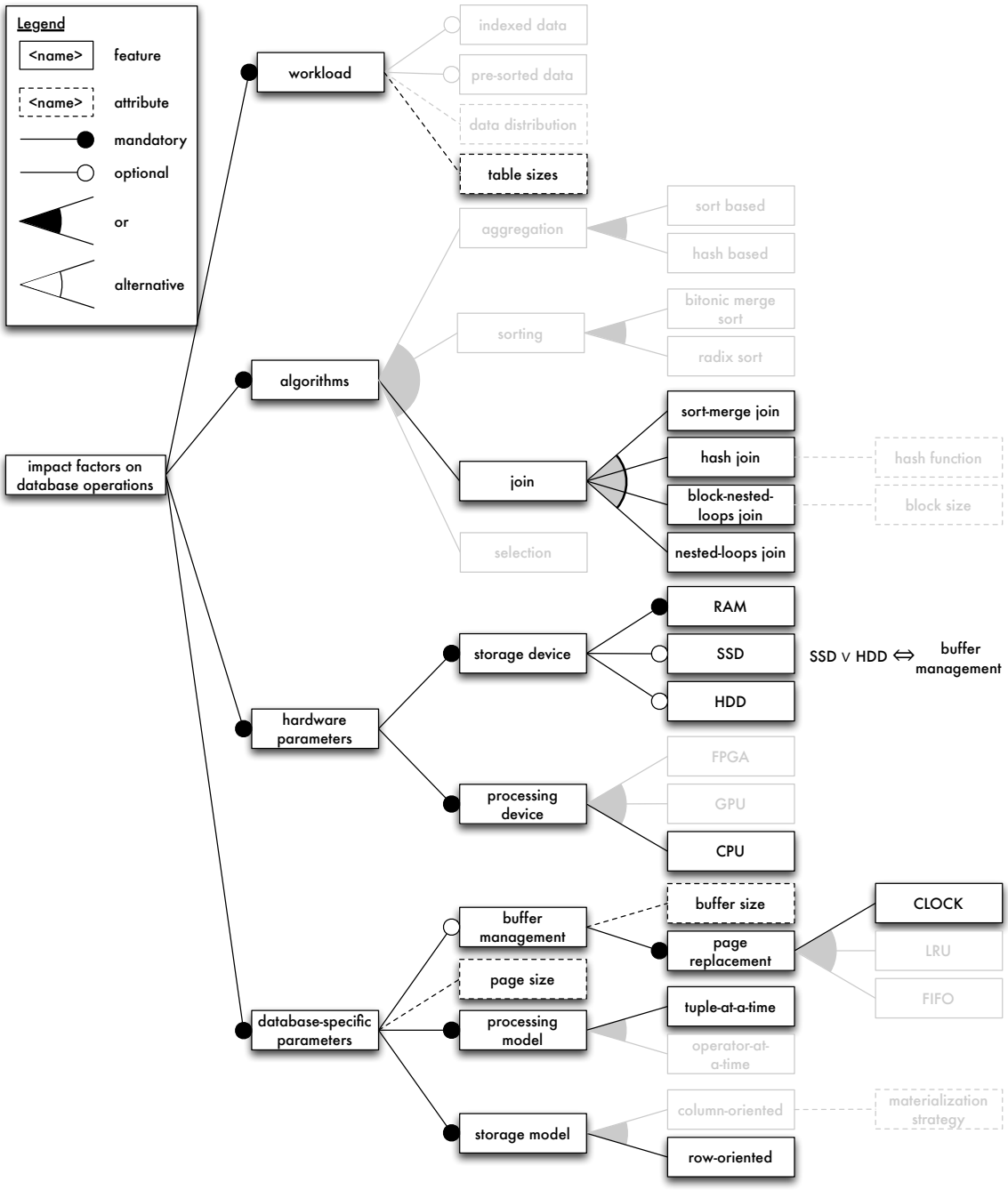


Figure 4.7: Reduced feature model of impact factors on performance of database operations.

down to an acceptable amount of variants to evaluate. However, further work includes evaluation of selection, sorting and aggregation.

To find the best join strategy depending on the impact factors, we consider nested-loops, block-nested-loops, hash, and sort-merge join, because we do not know which strategy performs best for a given scenario. Since we evaluate each join algorithm on its own instead of evaluating whole query plans with several joins, the join algorithms are represented in an *alternative* relation in the reduced feature model.

Considering additional attributes of the join algorithms, a comprehensive evaluation of different block sizes and hash functions would be necessary. However, this would lead each to an increase of at least factor 10 or more in the number of evaluated variants. Consequently, we have to exclude these two attributes, because a comprehensive evaluation of these two properties is not possible in the limited time of this work.

Workload

Data of our workload is taken from one of the most-often-used data-warehouse benchmark *TPC-H*[2]. It provides large volumes of data that are common in industrial use cases and business processes. The benchmark offers several tables and, hence, we are able to join tables of different sizes and find the best join strategy for them. All tables have the same distribution and are pre-order by their primary key. Since we consider only ad-hoc joins, we leave performance evaluation for indexed data open for future work.

4.3 Conclusion

In the first section of this chapter, we contribute a structured representation of the variant space. Our examination reveals that there are several hundred thousands of possible configurations and considering attributes, we end up having to execute an infinite number of evaluations. Nevertheless, the examination of the impact factors shows which parts in our implementation have to be exchangeable. Important points to be considered in the following are:

- Variable storage model
- Variable processing model
- Integration of new page replacement strategies
- Extensibility of algorithm pool

[2]http://www.tpc.org/tpch/

Because of the huge amount of possible configurations, we have to reduced the number of considered impact factors in order to achieve a feasible evaluation of selected impact factors. As a result, we contribute a specialized feature model to evaluate different storage devices for the traditional database architecture. This degrades the variant space to four possible storage configurations when considering only Boolean features. These four variants represent varying storage configurations, which are:

1. All data is stored on an HDD and we use the RAM as our buffer.
2. Our secondary memory device is the SSD and RAM is used for buffering.
3. All data is stored on an HDD. The SSD as well as the RAM are our two buffers, where RAM is the primary buffer and if it is too full, pages are written down to the SSD. In this case, we consider the SSD as extended buffer pool instead of as extended disk.
4. Our tables are stored in RAM and can be processed without the need of a buffer manager.

In addition to the four storage configurations, we include the four join algorithms nested-loops, block-nested loops, hash, and sort-merge join, which increases the number of variants to 16. However, the total number of evaluations that have to be executed depends on the selection of values for the attributes which are the buffer size, page size, as well as the chosen table sizes.

5. A Framework for Performance Evaluation of Database Operations

In this chapter, we describe how we implement a framework for performance measurements of database operations. As a first consideration, we argue why we favor a tool instead of cost models in the first section. In Section 5.2, we describe the architecture of our framework and state differences in our tool compared to the traditional five-level-schema architecture. With these sections, we underline the validity our performance evaluations using our tool.

Another focus of our work is to provide variability in our implementation. To implement the required variability regarding our variant space, at first, we use abstract classes to support variable implementations of the same class. The usage and characteristics of abstract classes is described in Section 5.3. However, some characteristics of abstract classes may influence the performance of our implementation and, consequently, we show an alternative implementation in Section 5.4.

5.1 Preliminary Considerations

Our goal is to determine the best algorithm of a database operation for a given system and workload. To test the hypotheses found in literature, we have to evaluate the performance of database operations. To this end, we decided to use a tool to measure performance instead of creating a cost model to estimate execution times. Important reasons for this decision are given in the following.

After deciding to implement a tool for performance evaluation of database operations, we have to choose a good programming language enabling us to do fine-granular performance measurements. For this, we reason our decision in Section 5.1.2.

5.1.1 Whether to Use a Tool or Cost Model

In general, there are two ways for a performance evaluation:

- On the one hand, after a comprehensive theoretical examination of all possible impact factors, costs are derived for the impact factors and a cost model is created. An important example is the cost model of Manegold et al. for access costs in in-memory databases [MBK02].

  Possible threats that make the model incomprehensive are missing impact factors and changing properties of considered impacting components. Especially, access time or processing power are weighted features in the cost model and the specific weight for a given system is not always easy to determine [SSH11].

- On the other hand, a software system can be developed which is executed on the given hardware measuring the performance of database operations under the specific workload. This strategy has already been used in QuEval[1] to evaluate the performance of index structures [GBS+12, SGS+13].

  As a result of our performance evaluation, we identify algorithm implementations that are superior to others and are able to formulate advices of algorithm usage under the specific workload. To prove the validity of our assumptions, a test on a complete database system has to be done.

Considering both possibilities, we decide to implement software that evaluates the database operations on a given hardware system, because of the following reasons:

1. Creating a cost model for varying impact factors is an error-prone task, because there are too many factors that have to be taken into account. The high amount of impact factors makes the cost model highly complex and if a critical factor was overseen, the validity of the cost model may be disproved. To discuss validity, performance tests have to be done under different configurations of the underlying hardware. However, as we have already stated, the number of possible configurations is extremely high, making prove of validity almost unachievable.

2. Another disadvantageous aspect of cost models is that some factors are not representable by variables. For instance, algorithm design is hard to cover in a cost model. In a tool, however, different implementations of an algorithm can be evaluated against each other.

3. Considering ideas for a reimplementation of specific database operations, users implement necessary algorithms and are able to test their implementation against others using our tool. With this, there is no need to understand a highly complex cost model and to map an implementation to variables in the cost model. Furthermore, unconsidered impact factors that arise from the given implementation would have to be included and weighted, which may introduce errors.

[1] http://QuEval.de

By implementing a tool for performance evaluation of database operations, we are able to test different algorithms and establish a framework in which every programmer may contribute. An open necessary decision is the programming language. To end this, we present our decision in the following section.

5.1.2 Choice of Programming Language

At first, we decide to use an imperative instead of a logic programming language, since our database operations are easier expressible in an imperative language. Furthermore, we rely on the object-oriented programming paradigm to use functionality, such as inheritance to provide necessary variability. Since we are executing performance measurements, we have to use a programming language that interferes as little as possible our processing. For instance, unpredictable garbage collection would hinder our performance measurements. Furthermore, for evaluating memory consumption of algorithms, we need a fine-granular memory management. As a consequence of the before-mentioned points, we use *C++* as programming language.

With the choice of the programming language, we also use functionality of several standard C++-libraries to implement our tool as well as database operations. Furthermore, we use some classes of the boost-library[2], which offers responsible implementations and is said to be included in the C++-standard soon. As a consequence, we have to rely on a proper implementation of used classes to reach a valid evaluation framework.

After presenting preliminary considerations that lead us to an implementation of an evaluation framework written in C++, the implementation itself is the focus of the following sections. Important points are the overall architecture of our framework as well as ways to provide variability in our implementation to support different configurations derived from Chapter 4.

5.2 Architecture of our Framework

For a comprehensive evaluation of database operations, our evaluation environment has to be known in particular to argue that our results are valid. Thus, we present the overall design of our implementation in this section and give an introduction to our framework. For this, we start with a short presentation of the workflow in our framework.

For a good implementation, which produces valid results, we derive our implementation from the traditional five-level-schema architecture. Nevertheless, since our goal is to evaluate the performance of database operations, we are able to define some assumptions that simplify our implementation in comparison to an implementation of a whole database management system. For instance, transaction management is not required, because it is not in the focus of this work. Since our simplifications may influence the performance, we discuss differences in our implementation to the five-level-schema architecture and also discuss the validity of our results considering the given limitations.

[2] http://www.boost.org

5.2.1 Workflow in our Framework

To execute an evaluation of database operation performance, we define the following workflow when using our framework. In the first step, necessary tables are created one after another, resulting in a sequential storage of tables on pages. For this, the schema of a table is defined in the data dictionary and then data can be inserted. We emphasize that our framework stores all data of one table on contiguous pages. With this, we have to store only the first page of the table to know where tuples are stored. We argue that instead of storing tables on contiguous pages, we could also use simple data structures such as *hash maps* to provide the same functionality with minimal overhead.

As the second step, different database operations can be executed on created tables and execution times are measured. For executing the operations, we assume that intermediate data structures, such as hash tables for the hash join, fit into RAM. Otherwise, these data structures would have to be written on pages and placed on disk. However, we argue that intermediate data structures fit well into RAM, because they do not store whole tables. Instead, data of few columns is stored causing little overhead.

At last, tables may be stored after the execution. With this, we can check whether two different algorithms of an operation produce the same result and, thus, validate the correctness of our implementation.

5.2.2 Simplification of the Five-Level-Schema Architecture

Considering the workflow of our framework and needed functionality, we are able to reduce the requirements of a traditional database management system in our case, because we measure the performance of database operations instead of several queries in a transactional context. In Figure 5.1, we depict the necessary components for our implementation and show unnecessary components or components whose functionality is already provided in gray color. In the following, we argue for our decisions and discuss the validity of results under the given simplifications.

Data System

In contrast to traditional database management systems, our framework does not require functionality of the data system. The data system is responsible for translating SQL-queries into an inner representation and their optimization. Since we measure the performance of database operations, formulating whole SQL-queries is not necessary to tell the system what to do. Consequently, parsing and optimizing complex SQL-queries is not supported in our system. Instead, we create tables and execute database operations by simply calling functions, which looses the comfort of SQL but provides comparable and distinctive results.

Access System

Considering the access system, we implement a data dictionary which holds all table definitions including lengths of columns and data types. For a first step, we are supporting 32-bit and 64-bit integers as well as the data type `char(`n`)` where n represents the

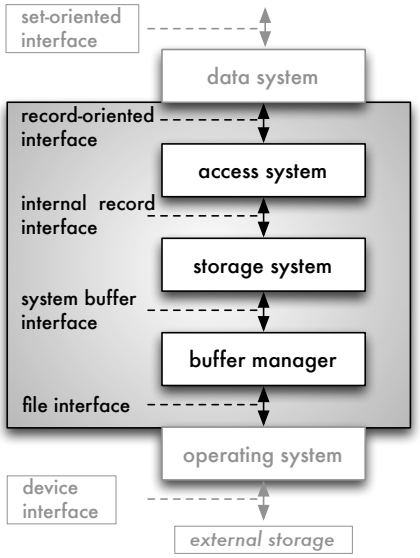


Figure 5.1: Implemented and unimplemented (shown in gray) components of the five-level-schema architecture.

number of chars of storable strings. Hence, supported data types have a fixed length, which is advantageous for our implementation. Since every row has a fixed length, storage locations can be computed using fixed offsets stored in the data dictionary. Extending our framework for data types with varying lengths is left open for future work.

Storage System

Our storage system has the task to map TIDs to pages. For this, it uses our assumption, that tables are sequentially stored on contiguous pages. Consequently, since we know the first page of the table and the size of one row from the data dictionary, we can easily compute the page where the tuple is located and its offset on the page.

Considering the storage of tuples on pages, we do not support updates or deletes. As a consequence, a mixture of tuples, which would occur under many updates and deletes, is not accomplishable. Nevertheless, we argue that such a mixture can almost completely be emulated in our framework by just inserting tuples in a mixed order.

Buffer Manager

Since our storage hierarchy may differ considering the resulting four configurations of the last chapter, we have to be able to support several buffer managers with different levels of buffer pools, e.g., buffering on SSD and RAM to bridge the memory gap. Our solutions for the variability in the buffer-manager functionality are covered in the following section.

In general, our buffer manager navigates through files and loads pages. Furthermore, if the buffer pool is full, it has to replace a page that was identified by our page replacement strategy. For this, we implement the CLOCK algorithm as page replacement strategy.

Operating System

For our processing, the framework interacts with the operating system, but does not implement operating-system functionality for itself. Important functionality, we use, is the file interface to get necessary page data. Furthermore, we rely on the storage management of the operating system in RAM, which provides memory for our created objects and data types.

To summarize, we implement a framework for emulating the workflow of database operations in a typical database management system. For this, we identify simplifications of a database management system and argued that they simplify our implementation without reducing necessary validity for our evaluation.

5.3 Providing Variability with Abstract Classes

In the last section, we identified variable parts that we have to support in our implementation. After previously presenting the overall components and functionality of our framework, we present how we provide the variability of the buffer manager, page replacement, and join strategies in our framework in this section.

To provide the variability, we have to be able to exchange the code when executing different configurations. For this, we decide to use *abstract classes* as an object-oriented paradigm to create different implementations which can be handled similarly.

An abstract class in C++ can be used to define an interface for classes that inherit the method signature. Subclasses have to implement needed functionality of inherited methods for the current configuration. The advantage is, when declaring classes, an abstract class can be initiated with any subclass at run-time. Thus, a uniform handling of different implementations is accomplished. A good example for abstract classes as interfaces for different implementations is given in the following for the buffer manager.

Introducing abstract classes in our implementation will cause additional computational overhead [DH96]. As a consequence, we also want to review another possibility and argue possible advantages and disadvantages of both approaches in Section 5.4.

5.3.1 Interface of Buffer Manager

Our implementation of the buffer manager has to be able to support different storage scenarios. In the first configuration, data is stored on SSD or HDD and pages are buffered in RAM. In this case, we argue that the buffer manager has to provide the same functionality for both devices, because only the storage location differs. The second configuration is an extension to the first one where we add another buffer to the system which uses the SSD. The third configuration that we consider is the in-memory case where all data is already stored in RAM and a buffer manager that has to replace pages is not necessary.

Code of Abstract Class `BufferManager`

For a variable implementation of the three storage configurations, we implement an abstract class `BufferManager` that defines necessary methods. Our code for the `BufferManager` is depicted in Listing 5.1.

```
1  class BufferManager{
2
3  protected:
4
5      pageContainer* pages;
6      unsigned int numberPages;
7
8  public:
9
10     virtual PagePtr getPage(unsigned int pageID, Table* table) = 0;
11     virtual PagePtr getNewPage(Table* table) = 0;
12     virtual void commit(Table* table) = 0;
13     virtual void clean() = 0;
14     virtual ~BufferManager(){};
15
16 };
```

Listing 5.1: Abstract class `BufferManager`.

Considering necessary methods and fields, a buffer manager needs a container that holds the pages which are buffered, which is called `pages` (cf. line 5), and a number of pages to identify new page IDs.

Since the buffer manager has to load pages, we define the method signature `getPage` in Line 10. This is a pure virtual method, because it does not contain any code in the abstract class and, consequently, should not be executed. Behavior of pure virtual methods should be implemented in derived classes. Further pure virtual methods are `getNewPage`, which creates a new page for the table, `commit`, which writes every page of the table that is in RAM down to persistent storage, and `clean`, which writes every page in the buffer down to persistent storage.

Additionally, we define the destructor ~`BufferManager` as virtual method, because it has to be able to be called when destroying a `BufferManager` object and if we do not define a constructor for the class, the standard constructor is used which is sufficient for our implementation.

Three Implementations of Buffer Manager Derived from `BufferManager`

To have a variable implementation, we have to create derived classes for the three beforementioned configurations and implement the pure virtual methods of the super class in the derived classes. In Figure 5.2, we depict the UML-diagram of the implementation of our different buffer managers.

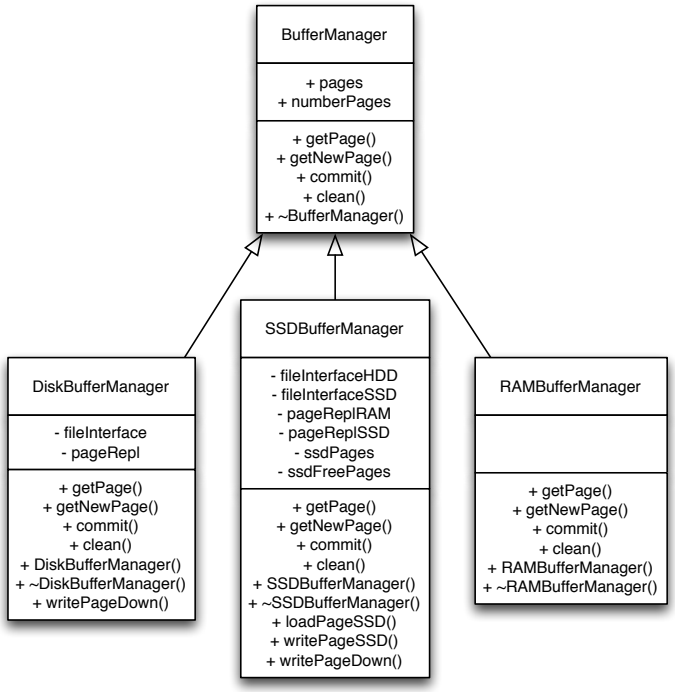


Figure 5.2: Buffer manager implementations for different storage devices ("+" denotes public class members and "−" private class members).

The `DiskBufferManager` introduces the member `fileInterface`, which enables access to SSD or HDD files, and `pageRepl` being a page replacement strategy for our buffer in RAM. Furthermore, it implements the defined methods and adds the method `writePageDown`, which writes pages to the storage device under the usage of the file interface.

Considering the `SSDBufferManager`, it has to manage two buffers, one buffering pages in RAM and another one managing pages on SSD. Consequently, we need two page replacement strategies, and an additional page container which manages page location on the SSD. Since we access data on SSD via the file interface, we have to create two file interfaces – one for the HDD and one for the SSD.

The easiest configuration to implement is the class `RAMBufferManager`, because many simplifications are possible. The methods `clean` and `commit` do not contain any code, because all pages stay in RAM. Furthermore, synchronization with persistent storage is not necessary, which makes a file interface obsolete.

How to Use Different Buffer Managers

After defining an abstract class `BufferManager` and deriving three different implementations depending on the used storage devices, we show exemplary code of its usage in Listing 5.2.

```
1 class Main{
2     ...
3     BufferManager* buff = new RAMBufferManager();
4     ...
5     buff->getPage(pageID, table);
6     ...
7 };
```

Listing 5.2: Initialization of `buff` as `RAMBufferManager`.

In Line 3, we show the initialization of a pointer to a `BufferManager` which is instantiated with the derived class `RAMBufferManager`. Since we can initialize `buff` with every derived class, we are able to handle different implementations uniformly. This offers the possibility to exchange the functionality of the BufferManager during runtime, but also introduces computational overhead, which we will explain by introducing static and dynamic object types in the following.

In C++, every object has a static and a dynamic type [Mey05]. The static type is the one, which is defined at compile-time. In our listing, the static type of `buff` is a pointer to `BufferManager`. However, the dynamic type differs from that, because we initialize it with a pointer to `RAMBufferManager`, which is the dynamic type of `buff`.

As we have seen, the dynamic type of the variable `buff` may change during processing. This influences the computational effort when executing virtual functions (cf. Line 5), because they depend on the dynamic type. If a virtual function is called, a lookup on a virtual function table becomes necessary to get the address of the function to be executed introducing an additional indirection [DH96]. As a consequence, we discuss alternatives in Section 5.4 to avoid abstract classes and their computational overhead to get more unbiased results.

5.3.2 Page Replacement Strategy Interface

Another variable part of our implementation is the page replacement strategy, because new strategies should be added in the future. At the moment, we implement the CLOCK-algorithm as page replacement strategy, as depicted in Figure 5.3. To have a comprehensive interface, we identified the following methods to be provided:

- The buffer size should be adjustable so that we are able to execute performance measurements under varying buffer sizes.
- Pages have to be added and removed from the strategy if a new page is loaded into buffer pool or replaced.
- If a page is accessed or modified, the page replacement strategy has to be notified, because it possibly influences the decision of the strategy, which page will be replaced next.
- Furthermore, it has to decide which page is replaced if the buffer is full.

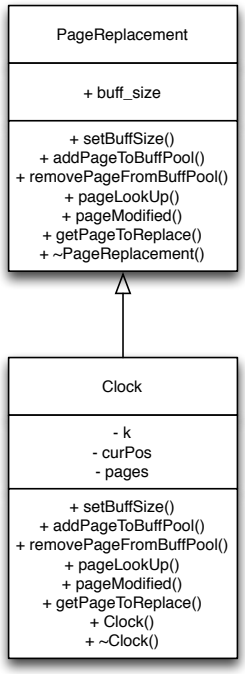


Figure 5.3: Interface for page replacement strategies.

The predefined methods are implemented for our page replacement strategy CLOCK, which works as follows. CLOCK stores a counter for every page, which is initially set to the given value k. If the page is accessed, its counter is incremented if it is smaller than k. If a page has to be replaced, the list of page counters is traversed in a cyclic way and every time the current counter is greater than 0, it is decremented. When an accessed counter is 0, the page is the returned candidate for replacement.

5.3.3 Implementation of Join Algorithms

The interface of our join algorithms is straight forward. Each algorithm has to provide a method to join two tables. This creates a joined table and persistently stores it on the storage devices. With this, we are able to test the validity of our join algorithm results.

For the performance evaluation, we will produce a join index using the method `joinIndex`. Our join index consists of pairs of TIDs that represent the matching tuples. As a consequence, we save computational effort for writing tables to disk. This is negligible, because every algorithm writes the same result table to disk.

As depicted in Figure 5.4, we derive the `NestedLoopsJoin`, `BlockNestedLoopsJoin`, `HashJoin`, and `SortMergeJoin` from our abstract class `JoinAlgorithm`. To provide the variability in our join algorithms, we add fields for block sizes in the block-nested-loops join and an exchangeable hash table implementation in the hash join.

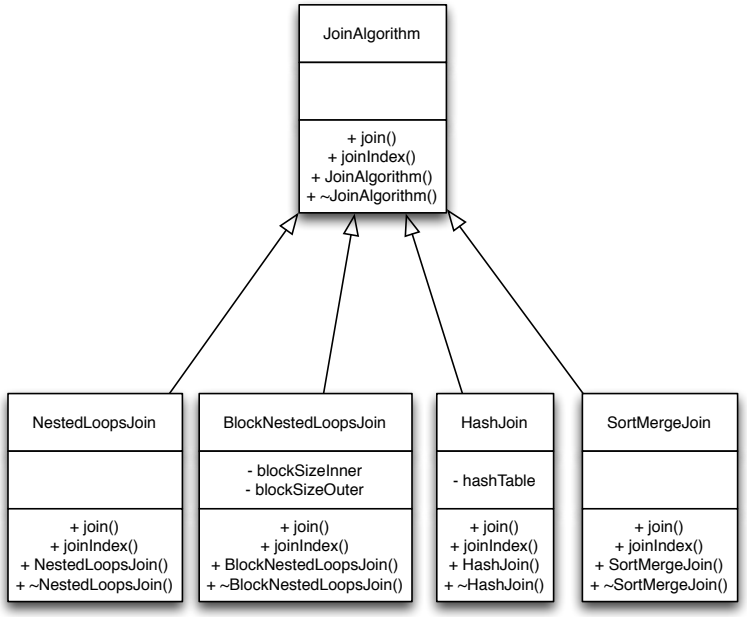


Figure 5.4: UML-diagram of join strategy implementations.

5.3.4 Variability in Access Using Cursors

Another important task is to allow different processing models. For this, we have to be able to access either single tuples one after another, or to process them all in one iteration. To allow access to an unbound number of tuples, we implement cursors for accessing the table content. The UML-diagram of different cursor implementations is shown in Figure 5.5.

Every cursor retrieves tuples from the table and passes them forward. Considering the `SingleTupleCursor`, one tuple after another is loaded and as long as the end of the table is not reached, the next tuple can be loaded. To allow loading multiple tuples in one step, cursors return a vector of tuples which has the length one for the `SingleTupleCursor` and a predefined length for the `BulkCursor`. With this, processing of multiple tuples is enabled, which can be used to execute operator-at-a-time processing.

5.3.5 Summary

In this section, we introduce abstract classes as an object-oriented possibility to provide variability in our implementation. Abstract classes define an interface that has to be implemented in inheriting classes. With this, different implementations of one class can be handled uniformly in our implementation. However, virtual functions of abstract classes introduce computational overhead and, thus, we review another possibility to provide variability in our implementation.

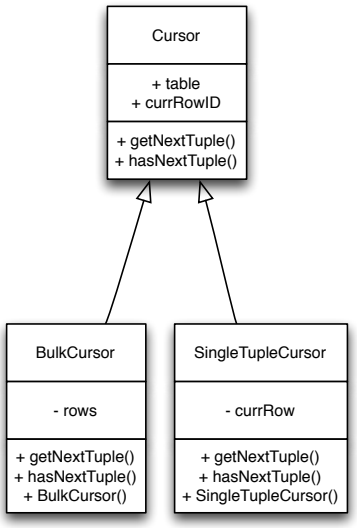


Figure 5.5: UML-diagram of different cursor implementations.

5.4 Providing Variability with Preprocessor Directives

Another way to support different implementations for classes and functions is to use preprocessor directives. As the name preprocessor directive implies, these statements are executed before the program is compiled. With this, we provide *compile-time variability* instead of run-time variability using abstract classes [ABKS13].

In the following, we present the syntax of preprocessor directives on the example of our configuration of different buffer managers. After that, we discuss the applicability of preprocessors in our example and present advantages and disadvantages.

5.4.1 Preprocessor Syntax

With preprocessors, we can define variables such as `INMEM`, `SSDBUFF`, or `HDD` to define different variants as shown in Listing 5.3. With "`#`" we introduce that the following content belongs to a preprocessor directive, which is processed by the preprocessor and will not be contained in the compiled code anymore. With `define`, we create a macro that is replaced with the value that follows the macro name.

For our implementation, we decide to use three macros. Normally, two variables would suffice to express four configurations, but for a better understanding of the code, we introduced three macros. In Listing 5.4, we present the instantiation of three different buffer managers.

With available preprocessor directives, we instantiate a `RAMBufferManager` iff `INMEM` is true. Otherwise, iff `SSDBUFF` is true, an `SSDBufferManager` is instantiated. However, iff none of these macros is true, the only left buffer manager is the `DiskBufferManager`.

```
1  class Main{
2  ...
3  // RAM or disk storage
4  #define INMEM false
5  // SSD as buffer?
6  #define SSDBUFF false
7  // HDD or SSD
8  #define HDD true
9  ...
10 };
```

Listing 5.3: Preprocessor directives for different configurations of the buffer manager.

```
1  class Main{
2      ...
3  #if INMEM
4      RAMBufferManager* buff = new RAMBufferManager();
5  #elif SSDBUFF
6      ...
7      SSDBufferManager* buff = new SSDBufferManager(pageRepl1,pageRepl2);
8  #else
9      ...
10     DiskBufferManager* buff = new DiskBufferManager(pageRepl);
11 #endif
12     ...
13 };
```

Listing 5.4: Defining different buffer managers.

Since preprocessor directives are used without brackets, we have to end the if-block with `#endif`.

Using preprocessor directives, we are able to introduce variability in our program. Only necessary code is passed to the compiler and, with this, we expect minimal computational overhead, as we discuss in the next section.

5.4.2 Expected Advantages and Known Drawbacks of Preprocessor Usage

Preprocessor directives are the state-of-the-art approach and have been being used for decades. Thus, their impact on the implementation and especially their understanding are well-documented. In this section, we want to discuss known benefits and drawbacks that we encountered while our implementation. For a detailed discussion of known impacts, we refer to Apel et al. [ABKS13]. Furthermore, we expect differences in the performance of our implementation when using preprocessor directives instead of abstract classes, which is discussed in the end of this section.

Known Advantages of Preprocessor Usage

An advantage of preprocessor directives is that they are easy to use and also well-known. We found much information about its usage and even our used programming IDE supported us in the implementation process with preprocessor directives. Thus, it is a good way to start with when introducing variability in a program. Furthermore, we are able to easily adapt our framework to use preprocessor directives instead of abstract classes, because preprocessor usage implies little preplanning. Another important advantage is that preprocessor directives allow us to vary programming artifacts of arbitrary granularity. We are able to include or exclude whole classes or just change the parameter of a function. An example for a fine-granular usage of preprocessor directives is shown in Listing 5.5

```
1  class CSVLoader{
2      ...
3  TablePtr loadCSV(char delimiter, DataDictionary* dict,
4  #if INMEM
5      RAMBufferManager*
6  #elif SSDBUFF
7      SSDBufferManager*
8  #else
9      DiskBufferManager*
10 #endif
11     bufferManager,
12     std::ifstream* openedFile);
13     ...
14 };
```

Listing 5.5: Fine-granular adaptation using preprocessor directives.

The class `CSVLoader` enables loading table data from a file in CSV-format into our framework. For this, it needs access to the buffer manager to place tuples one the specified pages. However, the buffer manager may change and, hence, the method signature of `loadCSV` has to change as well.

Disadvantages of Preprocessor Usage

However, there are some disadvantages when using preprocessor directives, which grant them the name *#ifdef hell* [FKA+13]. One of them is visible in Listing 5.5, because a fine-granular adaptation implies an overhead of code, which makes it more confusing. With the usage of preprocessor directives, the declaration of the method `loadCSV` changes from the extent of a single line to being scattered over several lines.

Furthermore, we had to adapt code distributed on several classes. A desirable property would be a clear *separation of concerns* [ABKS13] in our implementation. This means, code that belongs to one feature is located in one programming entity. However, this is not possible by just using preprocessor directives, because the directives belonging to one feature are situated in different classes.

Expected Differences in Performance Using Preprocessor Directives

As we have described above, abstract classes introduce an additional lookup on the virtual function table to find the code to be processed while run-time. Using preprocessor directives, we want to avoid this additional lookup and, hopefully, save computational effort. As a consequence, the performance of database operations may vary depending on the two framework implementations.

Designing our framework using preprocessor directives, the footprint of our variants should decrease. This is because unnecessary code, which was used to provide variability considering the usage of abstract classes, is filtered out before compilation and, thus, the footprint of the program should be shrunk. As a consequence, the performance of the framework should increase even more, because code of our framework fits better into cache and, hence, the possibility for a cache miss should reduce. Nevertheless, the given assumptions have to be proven in our evaluation to assure their validity.

5.4.3 Conclusion

To conclude, preprocessor directives are the state-of-the-art approach for introducing variability in C++-programs, because preprocessor directives are easy to use and offer adaptations of arbitrary extents. However, the code of one feature is often scattered over several classes and using preprocessor directives may reduce the readability of the code, because of the possibility for fine-granular adaptations. Furthermore, using preprocessor directives, we hope to be able to reduce the run-time overhead of abstract classes. However, we have to prove our assumptions in the evaluation.

5.5 Summary of our Implementation

In this chapter, we started with presenting arguments for an implementation of a framework to evaluate the performance of database operations. After that, we presented the overall design of our implementation. For this, we described the architecture and workflow of our framework. Furthermore, we present simplifications of our implementation compared to traditional database management systems. With this, we can state the validity of our results that are presented in the following chapter.

Another contribution of this chapter is to state possible extensions to our framework. Using abstract classes, we can extend the buffer manager, implement new page replacement strategies and join algorithms. In addition, we are able to access data tuple-wise or to load a whole table using cursors, which allows us to emulate different processing models.

However, we argue that the object-oriented way of implementing variability using abstract classes may introduce a performance overhead and, thus, we propose to use preprocessor directives as the state-of-the-art approach to introduce variability in C++. To discuss which of these approaches offers the best performance and introduces the smallest amount of computational overhead, we evaluate both approaches in the next chapter.

6. Evaluation

In this chapter, we present results of our tests that we have executed using our framework to evaluate the performance of join algorithms under varying hardware. As described in Chapter 4, we limit our considerations to different join strategies of tables with varying size under different storage configurations. For this, we will start to evaluate the traditional storage concept consisting of HDD as persistent storage and a buffer in RAM with varying capacity and varying page sizes. After that, we emphasize differences to other configurations of the storage system.

Another important fact to be evaluated is whether using preprocessor directives instead of abstract classes is beneficial for the performance of join algorithms. For this, we execute our experiments on the modified framework with preprocessor annotations.

6.1 Configuration of Evaluation

To have a comprehensive overview on our evaluation, we present defined parameters and impact factors in the following. For this, we will start by presenting hardware factors of our test machine and continue by introducing the workload of our tests.

6.1.1 Test Environment

We execute our performance evaluation on a machine with an Intel(R) Core(TM) i5-2500K as processing device. This is a quad-core with 3.30GHz processing power having 64KB L1-cache and 256KB L2-cache per core as well as 6MB L3-cache.

To provide different devices of the memory hierarchy, our test machine has an HDD, SSD, and RAM installed. The HDD is a Seagate ST380011A Barracuda with average latencies of 8.5ms. Our SSD is an ADATA Premiere Pro SP900 with 128GB storage capacity, which offers us average access times around 0.1ms at 550/520MB/s read and write performance respectively and, thus, being in the upper performance segment of

todays SSDs. We installed 2x4GB DDR3-RAM running on 1333MHz with a clock timing of 9-9-9-24.

As operating system, we installed the 64-bit version of Ubuntu 13.04, which offers us stability and a good performance for our tests. To provide robust results, we execute each performance measurement 20 times and compute the mean value out of it. To assure even more robustness, we use a two-sided gamma trimming approach to remove outliers, which means we filter out the two slowest and the two fastest response times.

Since we want to measure the performance of one database operation, we want to eliminate the possibility that data or functions are already cached. For this, we execute all our implemented join algorithms per run instead of executing one algorithm 20 times and then continue with the next one. We argue that this is more relevant, because in a database scenario with many parallel transactions, we cannot assume that data or functions are already cached in the cache hierarchy.

6.1.2 Workload Characteristics

For the following discussion of our results, it is important to know the workload in particular. Especially, the distribution of tuples on the pages is important to explain dips in our performance diagrams. Our workload consists of five tables taken from the TPC-H benchmark, which we adapt to our supported data types. Important properties of the used tables, namely `region, nation, supplier, customer,` and `lineitem` table are summarized in Table 6.1.

	`region` Table	`nation` Table	`supplier` Table	`customer` Table	`lineitem` Table
Tuples	4	25	100	1500	60175
Columns	3	4	7	8	16
Size of one Tuple in Byte	181	185	197	223	159

Table 6.1: Summary of table properties.

Since tables are stored on pages, it is even more important to know how many pages have to be accessed for an operation. Page accesses are directly correlated to the response time of an operation. Furthermore, considering a buffer manager, it is important to know how much content of a table can be held in the buffer, because it has a limited amount of capacity of pages. To present an overview, we list the amount of occupied pages per table depending on the page size in Table 6.2. For our evaluation, we choose page sizes of 4, 8, and 16KB, because these are typical values used in common database management systems, such as PostgreSQL[1].

For evaluating the performance of join algorithms, we execute joins of four different table pairs. Considering our five tables, the TPC-H benchmark provides joins between

[1] http://www.postgresql.org/

Occupied Pages at	`region` Table	`nation` Table	`supplier` Table	`customer` Table	`lineitem` Table
4KB page size	1	2	5	84	2407
8KB page size	1	1	3	42	1180
16KB page size	1	1	2	21	585

Table 6.2: Pages per table under different page sizes.

`nation` and `region` table, `nation` and `supplier` table, `nation` and `customer` table as well as a join between `supplier` and `lineitem` table.

6.1.3 Expected Results

In this work, we want to emphasize and evaluate the impact of hardware on the performance of database operations. Consequently, our evaluation concentrates on the join performance differences under varying hardware. Generally, we want to find cases where the superiority of one join algorithm changes because of changed hardware and based on our literature review, we derive the following hypotheses:

- In contrast to the HDD, SDDs give advantage to random reads. Thus, join algorithms causing random reads should have a better performance using SSDs instead of HDDs.
- Compared to the configuration with the HDD as persistent storage device, the performance of database operations should increase when using the SSD as additional buffer, because it may close the gap in the memory hierarchy.
- Finally, the best performance results are to be achieved when using the in-memory variant.
- Furthermore, we expect that the object-oriented approach using many abstract classes incurs performance penalties compared to an implementation using preprocessor directives.

In the following, we will start by presenting our results for the traditional storage configuration consisting of HDD and RAM as the basis of our evaluation. With this, we are able to compare the remaining configurations to the traditional configuration and present differences between the storage configurations.

6.2 Performance Comparison for HDD as Storage Device

In this section, we are presenting numbers that we have gathered for evaluating the performance of different join strategies. For this, we executed the nested-loops, block-nested-loops, hash, and sort-merge join for the five tables on different storage configurations, buffer and page sizes. For a detailed view on our collected data, we refer to the Appendix Chapter A.

To present our results, we start with the evaluation of the traditional database configuration where all data is stored on the HDD and RAM is used as buffer. For comparing the performance of join strategies, we execute the four strategies on four joins under different buffer pool sizes and measure their response time. In Figure 6.1, we depict the average of measured response times of join algorithms for each pair of joined tables at 4KB page size with increasing buffer sizes.

For performance evaluation under varying buffer pool sizes, we measure the response time with 10 to 90 percent of the needed pages to fit into buffer pool. To calculate the number of needed pages and, thus, the buffer pool size, we refer to Table 6.2 presenting pages per table.

6.2.1 Join Performance When Joining `nation` and `region` Table

Regarding the performance of different join strategies, the join of the `nation` and `region` table is an interesting case, because the best join algorithm differs in this case depending on the available RAM. In Figure 6.1.(a), three stairs are visible:

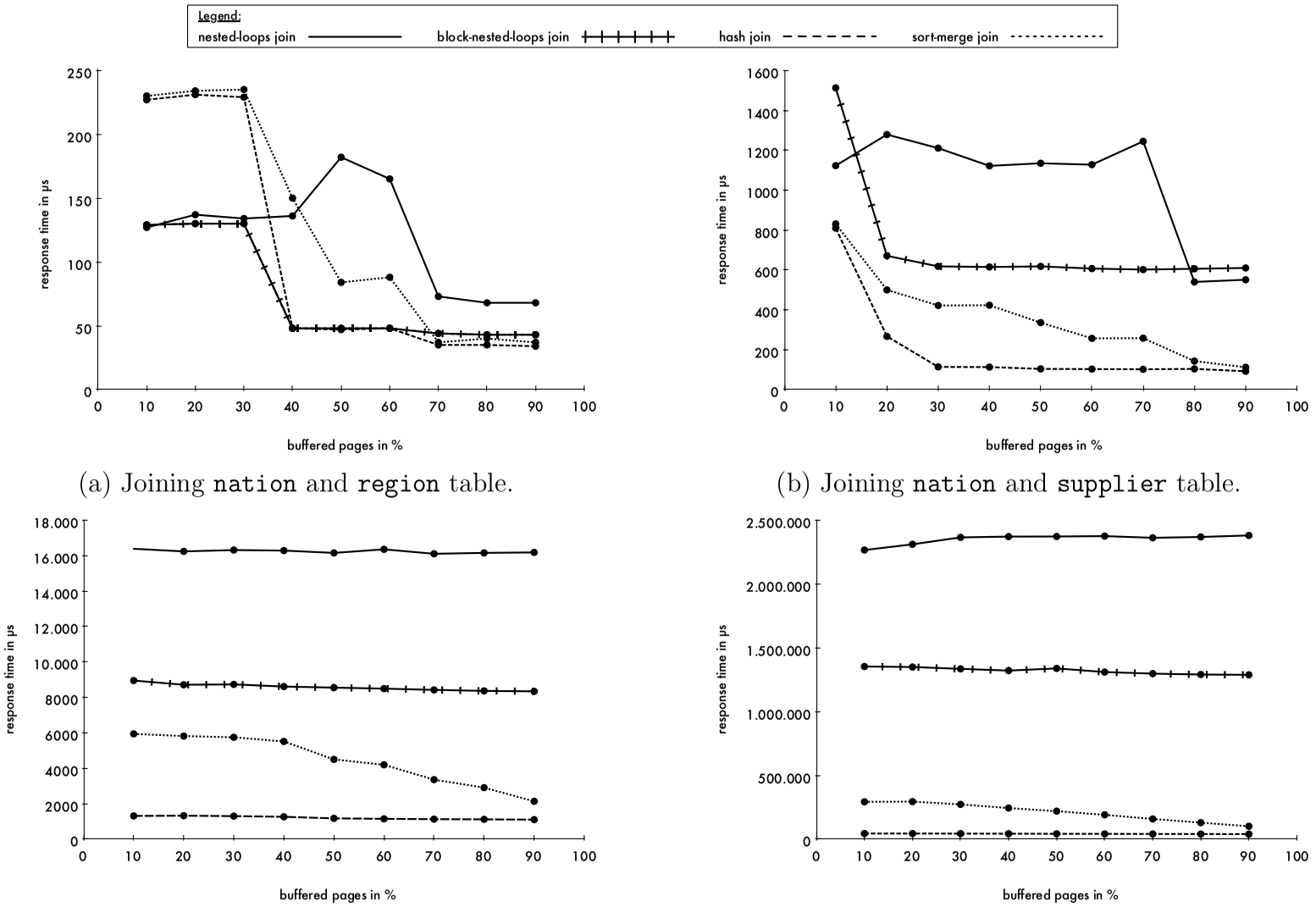
1. 10% – 30%: Since only one of three needed pages fits into RAM, each time a new tuple is read for comparison, the buffered page has to be replaced. At this configuration, the nested-loops and block-nested-loops join are superior, because they cause less page misses. For each tuple of the `region` table (4 in numbers), the two pages of the `nation` table are loaded.

   In contrast, the hash join loads the `region` table and constructs the hash table out of it. Then, for each tuple of the `nation` table (25 in numbers), the corresponding tuple of the `region` table is loaded which is located on the same side as the others. Consequently, the hash join cannot benefit from the hash table, because hashed values are located on the same side.

   Considering the sort-merge join, we are faced with the same problem. Both strategies, hash join and sort-merge join, are constructed for reducing the amount of loaded pages. However, this does only work, when (1) enough RAM is available and (2) page accesses are reduced because of the known storage location.

2. 40% – 60%: Here, two pages fit into RAM, which causes the same amount of cache misses for the nested-loops join, because still every page of the `nation` table is loaded. However, this configuration causes more computational effort for the nested-loops join, because the page replacement strategy has to traverse a list of two pages to find the page to be replaced.

   The sort-merge join improves its performance, because after reading both tables in, one page of each table can be held in RAM for comparison. Nevertheless, the sort-merge join is still not the best join strategy at this configuration, because the random access during comparison phase causes more page replacements than ,e.g., the hash join or block-nested-loops join, which benefit from the available RAM, having four page misses both.



(a) Joining `nation` and `region` table.

(b) Joining `nation` and `supplier` table.

(c) Joining `nation` and `customer` table.

(d) Joining `supplier` and `lineitem` table.

Figure 6.1: Response time of join algorithms on different tables.

3. 70% – 90%: All pages fit into RAM and except for loading pages, no access to secondary memory is needed. This configuration is similar to the in-memory case. The nested-loops join has the worst performance because of the computational overhead for comparing each tuple of one table with each tuple of the other. The block-nested-loops join proves his superiority to the nested-loops join, which is the result of the reduced I/O-overhead (cf. Section 2.3.3).

   The sort-merge join is the second fastest algorithm, because the sort phase needs additional computational effort. This join algorithm is only beaten by the hash join, which benefits the most from the data being in RAM.

As we can see, the best algorithm for joining small tables depends on the given RAM. This phenomenon is even more severely, when the page sizes increase, because tuples are placed on less pages. At 8KB page size, every table consumes one page and, as a consequence, there are just two steps in the performance. We summarize the average response times for the join algorithms at 8KB in Table 6.3.

Average Response Time at	Nested-Loops Join	Block-Nested-Loops Join	Hash Join	Sort-Merge Join
1 page in RAM	118	90	258	255
2 pages in RAM	55	41	34	43

Table 6.3: Average response time of join algorithms joining `region` and `nation` table with 8KB page size.

6.2.2 Join Performance When Joining Bigger Tables

In contrast to joins of small tables, the overhead for page accesses is dominantly impacting the performance of join algorithms. Since the nested-loops and the block-nested-loops join represent the brute-force approach, which accesses every page of the second table per tuple of the first table, they cause the most page accesses compared to the hash join and the sort-merge join.

However, the hash join and sort-merge join benefit from their selective page access in the second phase of the algorithm. This results in a big advantage for these two strategies, because significantly fewer pages are accessed and especially in a system where disk access is the bottleneck, reducing disk accesses is a beneficial strategy.

Our results indicate that, if joined tables take several pages, the hash join is the best join algorithm and is superior to every join algorithm that we implemented. Nevertheless, the sort-merge join offers good performance, too, and improves its response times the more RAM is available. Considering pre-sorted data, it is plausible, that the sort-merge join may have a better performance compared to the hash join.

6.3 Comparison for SSD as Storage Device

For comparing the performance of the join strategy on the SSD, we executed all experiments done on the HDD again using the SSD as storage location. While HDDs are good in performing sequential reads and suffer from random reads, an SSD beats the HDD in performing random reads, as we have already stated in Section 3.3.

Regarding differences in access times between HDD and SSD, join algorithms should be favored on the SSD that incur a high amount of random reads. Considering our four join algorithms, we expect the following results:

- The nested-loops join and block-nested-loops join rely on sequential reads and should not benefit significantly from SSD usage.
- However, the hash join and sort-merge join cause random reads in the second phase of their execution and, thus, benefit from the usage of the SSD.

In the second phase of the hash join, the second table is read sequentially and on the hashed table, random reads are executed. The sort-merge join causes even more random reads, because in the second phase, tables are traversed in the order of the join attribute values, which may lead to random reads, if the tables are not sorted on the join attribute. Consequently, we expect performance differences for the hash join and sort-merge join using SSDs instead of HDDs.

After executing the performance experiments on the SSD, we compared our results with the data gathered from the HDD configuration. However, there had been almost no significant difference in the corresponding response times of both implementations. Solely, the sort-merge join at 16KB page size shows steadily improved performance values of around 1%. Nevertheless, our results do not match our expectations and, thus, we will discuss, which impact factors may be changed to get more meaningful results for future evaluations.

6.3.1 Possible Impact Factors Distorting our Results

Since our assumptions mismatch the measured data, we have to find explanations for our gathered data. We identify four possible reasons that may have led to different performance results than we expected when using SSDs. These four are: our joins cause to less random reads, or too many write operations, the page size is not optimal, or the file is already cached. We discuss these four impact factors and present possible solutions in the following section.

Too Less Random Reads

One possibility that the SSD results have not improved for the hash and sort-merge join is that our workload implies to less random reads. For the hash-join, we execute random reads on the hashed table, which is the smaller table, because it was proposed to hash

the smaller table to reduce memory consumption [ME92]. However, the smaller table, which is either the `nation` table or the `supplier` table, takes between one and five pages depending on the page size. Consequently, it is very likely that the performance benefit for the hash join is too little in our scenario and, hence, cannot be separated from the standard deviation.

Considering the sort-merge join, data is read in the order of the join attribute values. As we have already said, one of the joined tables is already stored in a sorted way. Thus, it will be sequentially read when the join is executed reducing the amount of random reads.

To summarize, the used benchmark and the strategies may reduce the amount of random reads and, thus, the high potential of SSDs may not be exploited in our evaluation.

Too Many Write Operations

Another property of SSDs that may hinder their performance is the write latency of SSDs. As we review in Section 3.3, critical operations on SSDs are deleting of blocks to get free pages to write. In our experiments, every time the buffer is full and an additional page is loaded, one page in the buffer pool has to be written down to the SSD. These writes may decrease the performance of database operations if they are frequently executed.

Page Size not Optimal

Another possible impact factor that diminishes the superiority of SSDs as persistent storage device for database operation may be the chosen page sizes. For our evaluation, we choose 4, 8, and 16KB as page sizes. It is possible, that at the given granularity of access, the SSD configuration has a similar performance as the HDD configuration and differences would occur when page sizes are varied. As a consequence, further experiments are necessary to give a comprehensive answer considering the optimal page size for SSDs.

Database File is Cached

Our implementation relies on the functionality of the operating system, which provides access to resources, such as main memory and disk access. However, considering our scenario, the operating system may try to improve our performance by caching necessary data. Especially access to our database file may be improved by holding part of or even the whole file in RAM and propagating changes to the disk in an asynchronous way.

By caching the database file, the operating system improves the performance of our program, but also obscures the access gap. Another cache that may distort our results is installed on the hard disk itself. This cache buffers read and write request to reduce accesses to the hard disk. This may also lead to a better performance of the HDD compared to the SSD than we expected. As a consequence, we have to execute our experiments with more data, so that used caches overflow, which should favor the SSD as persistent storage devices.

6.3.2 Possible Solutions for Distorting Impact Factors

Since we cannot clarify which reason caused the mismatch between our assumptions and the measured performance of our join algorithms without further tests, we want to discuss here what has to be done in future work.

Increase Table Size

One possibility to increase random reads and stop caching effects is to increase the table size to amounts that are not cacheable for the operating system or hard disk. With this, we decrease the impact of caching in our performance evaluation and response times are dominated by real disk access.

However, increased table sizes also lead to highly increased response times, especially considering nested-loops join and block-nested-loops join. Thus, we have to leave this open for future work, because of our limited time.

Prevent Operating System From Caching by System Call

A possibility to stop caching effects of the operating system is provided by the Ubuntu operating system itself. When executing `echo 3 > /proc/sys/vm/drop_caches` on the command line tool, we are able to tell the Linux kernel to write all cached data down to disk again. This is an easy way to eliminate cache effects caused by the operating system, because C++ offers us the possibility to implement such system calls in our framework.

Nevertheless, when we tried to execute our experiments again with the included system call after accessing the database file, we were faced with enormous run times of the experiments. This arises from the fact that even the operating system itself needs cached data to work properly and fast enough. As a consequence, computational potential is used to resume the operating system functionality instead of executing our performance evaluation. Thus, we cannot use this system call to prevent caching, because it leads to false results.

Exhausting Available RAM

The problem of caching is that free memory is used to hold data that may be valuable for further processing even if we do not want it to be cached. But what if there is no space left to cache any data?

Considering our scenario, we would have to write a program that allocates remaining memory so that it cannot be used for caching our database file. Furthermore, the program should frequently access its data, because else the operating system may swap out unused data to disk in order to gain free memory for caching. The implementation of such a tool is open for future work.

Store Table Data Several Times

Another way to stop caching effects, is to blow up the size of the database file, so that it cannot be cached anymore. This could be done by storing the accessed table several times. With this, the database file can take an arbitrary size and, thus, is hard to cache. Additionally, to hinder the system from caching parts of the database file, we also have to access different replicas of the table. Considering our design, this is easy to implement, because we know the size of a table and how many tables are stored. Thus, storage locations of tuples in the replicas are easily computable.

However, accessing different replicas causes a random read every time which penalizes a HDD in contrast to the SSD and will also distort expected measurements. As a consequence, this strategy is only usable for selected algorithms that imply random reads by definition.

6.3.3 Summary

To summarize, we are not able to evidence that some algorithms, especially the hash and sort-merge join, benefit from SSDs as persistent storage devices. In this section, we described possible factors that may have distorted expected performance benefits and also present possible approaches using our framework to avoid these factors. However, since our performance tests would have to be repeated and we expect higher response for the algorithms, most of these possibilities have to be evaluated in future work, because of the limited time of this thesis.

6.4 Comparison for SSD as Extended Buffer Pool

Another configuration of storage devices includes the SSD as buffer pool to bridge the access gap between HDD and RAM. For this configuration, we firstly present some assumptions according to expected performance characteristics and, after that, we show if our expectations are confirmed by our gathered data.

6.4.1 Performance Assumptions

Considering the design of our implementation to use the SSD as additional buffer, we are able to formulate a few assumptions on the expected performance of join algorithms under different criteria. On the one hand, a high amount of RAM should diminish the performance gain when using SSDs to extended the buffer pool and, on the other hand, the number of reused pages determines how beneficial an additional buffer pool on the SSD is.

High Amounts of RAM Diminishes Benefit of SSD as Extended Buffer Pool

The implementation of the SSD buffer pool incurs additional overhead. First, an additional data structure is needed holding information of pages buffered in RAM. Second, if a page has been loaded from HDD and both buffers are full, two page replacements

have to be executed to buffer the new page. As a consequence, if enough RAM is available, the need for an SSD decreases, because the computational overhead is not justified.

With this in mind, we expect that at little amount of RAM available, performance of join algorithms increases compared to the HDD configuration. However, when the amount of available RAM increases, the gained speedups will decrease and join performance may even get worse than the corresponding join performance under the HDD configuration.

Page Reuse Influences Performance

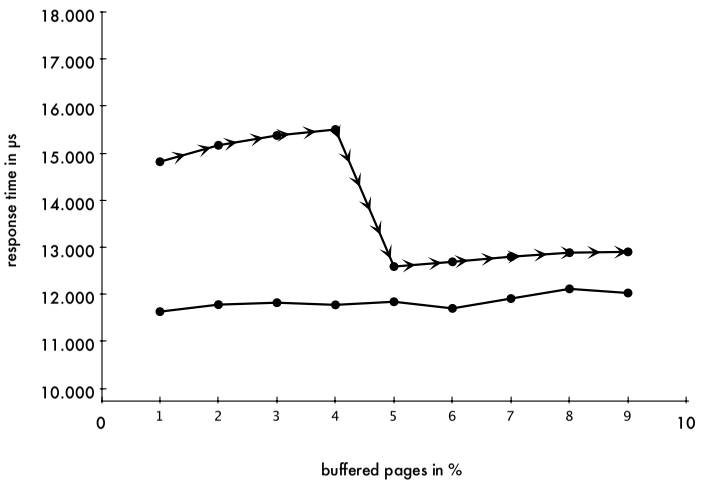
Having in mind that using the SSD as buffer pool implies computational overhead for buffering pages, we have to be sure that the usage pattern of our algorithm reuses buffered pages in an appropriate amount. If pages are buffered on the SSD, but not reused anymore, the computational overhead for holding them buffered on the SSD is not justified.

Considering our join algorithms, especially, the nested-loops as well as the block-nested-loops join access all pages per tuple. Consequently, if all pages do not fit en bloc into the buffers, reuse will not occur in this storage configuration and a benefit will not be achieved. Probably, hash and sort-merge join will benefit more from the SSD as extended buffer pool, because they access less pages and a reuse of already cached pages has a higher probability.

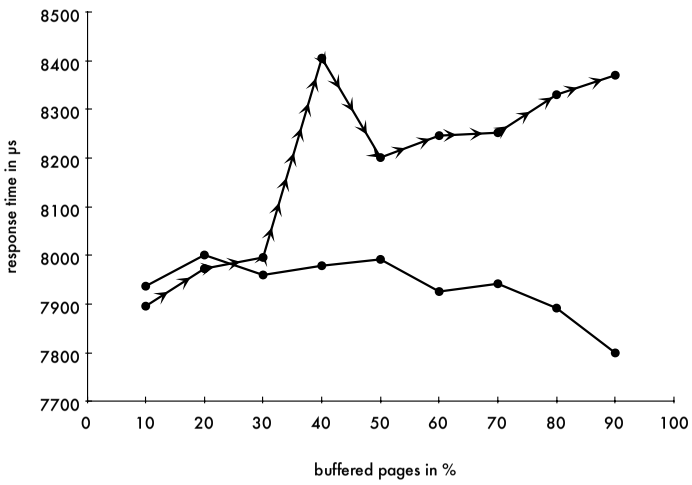
6.4.2 Performance Comparison between the Traditional Configuration and the Extended Buffer Pool

To make decisions, whether reached performance is an improvement to the traditional storage configuration, we have to compare measured response times under the two configurations. We choose the join of the `nation` and `customer` table, because it represents a good scenario and the probability of reusing pages at little amount of RAM is good. Since we are able to configure two buffer sizes when using SSD as buffer — one for the buffer in RAM and one for the SSD buffer pool — we have to find a way to compare both configurations. For this scenario, we observe that response times are very stable under different sizes of the SSD buffer pool and, thus, we decide to compare the response time under 50% buffer pool on SSD. The corresponding diagrams are depicted in Figure 6.2. Lines with arrows display join performance under the SSD as extended buffer pool configuration. The performance diagrams for the nested-loops and block-nested-loops join show that response time does not improve significantly using an SSD as additional buffer. This is caused by the cache-unfriendly access pattern of these both strategies, although the block-nested-loops shows slightly improved performance of around 0.5% for 10 and 20% available RAM.

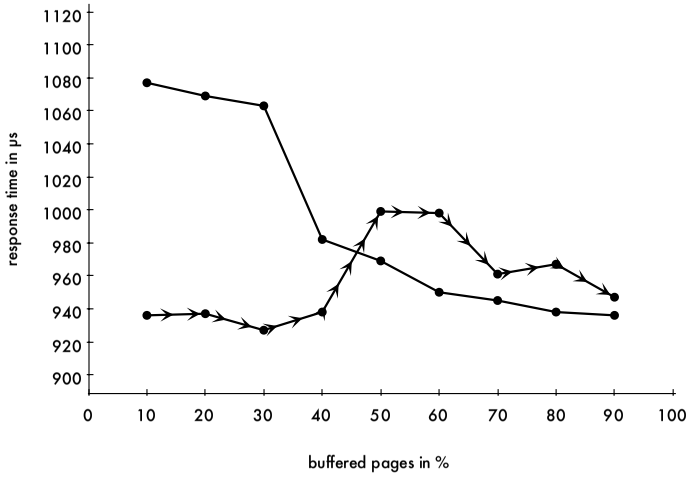
Furthermore, we observe speedups for the hash join and sort-merge join on our SSD as extended buffer pool configuration compared to the HDD configuration. However, the performance advantages of the hash join are present up to 40% of available RAM. If



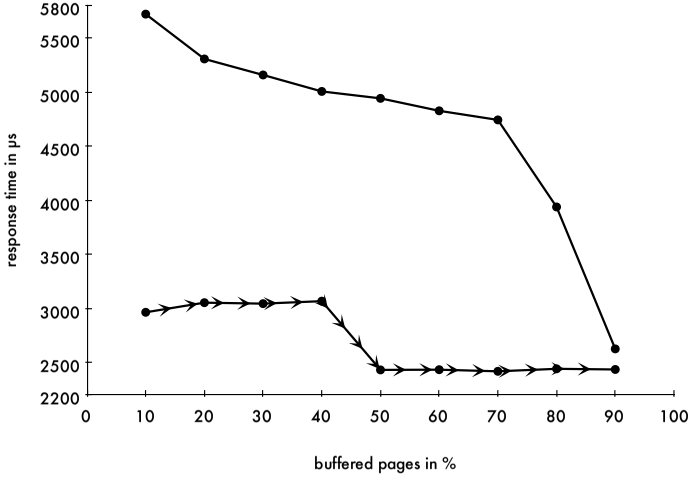
(a) Nested-loops join



(b) Block-nested-loops join



(c) Hash join



(d) Sort-merge join

Figure 6.2: Response time of join algorithms when joining `nation` and `customer` table under the traditional storage configuration of HDD and RAM (normal lines) and the SSD as extended buffer pool configuration (lines with arrows).

this border is passed, the computational overhead of two buffer pools is not justifiable and the traditional approach works better. Even considering the performance of the sort-merge join, speedups decrease with increasing amount of available RAM.

6.4.3 Performance Under Varying SSD Buffer Pool Size

Considering our gathered data, we observe an abnormal behavior of response times under varying buffer pool size if the available RAM is limited to not more than 30% RAM. Normally, response times are either relatively stable with increasing SSD buffer size or even improve. However, in some case, with increasing SSD buffer sizes, response times first improve and at a predefined border, they get worse again. We depict gathered data of the described scenario in Figure 6.3 where the `nation` and `region` table is joined.

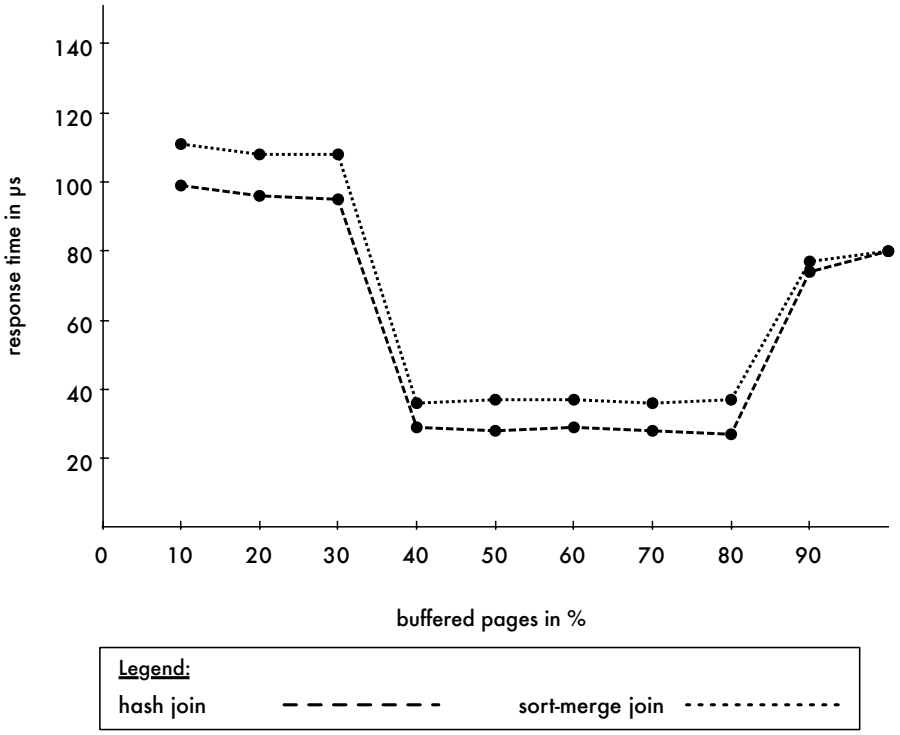


Figure 6.3: Hash join and sort-merge join under varying SSD buffer size for 10% available RAM buffer size.

As the curves for the response time indicate, there are special values of SSD buffer size that are more advantageous than others. At the given scenario with little RAM, it is advantageous to buffer more than 30% of the pages on the SSD to gain speedups. However, if more than 80% of the pages are buffered, the performance decreases again. This performance decrease arises, because storing a high amount of pages in the buffer pool implies bigger data structures and computational overhead to provide buffer functionality, which is not sufficiently compensated by performance gain of using SSDs.

To summarize, our performance comparisons show improved performance using an SSD as additional buffer, if pages are reused and the RAM is limited. In fact, the hash join and sort-merge join benefit from the usage of the SSD as additional buffer. Furthermore, in our experiments, we found out that it is only beneficial to use SSDs for buffering if the available amount of RAM can hold at maximum 40% of the data or less.

6.5 Comparison for In-Memory Configuration

The last configuration, we want to evaluate is the configuration where all data is stored in RAM. It is obvious, that this configuration has to be the fastest, because there is no impact from slower persistent storage devices. However, the memory wall represents the bottleneck to the caches here and, thus, we hope to achieve even faster response times with varying page sizes, because caches may be used more efficiently.

The first experiment is done with a page size of 4KB and measured averages of response times are presented in Table 6.4. While the hash join is between 23 and 44% faster than its competitors for the join of the `region` and `nation` table, it widens the performance advantage to up to 97% for the join of the `supplier` and `lineitem` table. As a consequence, the hash join is the best join strategy of our four given implementations.

Average Response Time	Nested-Loops Join	Block-Nested-Loops Join	Hash Join	Sort-Merge Join
`nation` & `region`	40	38	23	32
`nation` & `supplier`	571	565	74	104
`nation` & `customer`	8545	8503	830	1227
`supplier` & `lineitem`	1344499	1331276	33914	63643

Table 6.4: Average response time of join algorithms for the in-memory case at 4KB.

Considering response times for different page sizes, we do not discover a significant or steady improvement of performance. This may have the reason that either we execute our experiments on too less data, the page sizes are not well chosen to fit caches, or our algorithms are not designed in a cache friendly way.

As a consequence for future work, we have to examine cache misses when joining bigger tables under varying page sizes to get an idea what page sizes favor join algorithms the most. Furthermore, we have to examine our algorithm design and its incurring cache misses to improve their in-memory performance.

6.6 Influence of Preprocessor Usage

In the previous chapter, we have already emphasized that an implementation using abstract classes may introduce a performance overhead because of additional required lookups on virtual function tables. To reduce this overhead, we use preprocessor directives to implement variability in our framework. With this, we want to improve

the performance and footprint of our framework, which also should improve response times of join algorithms. In the following, we present whether the usage of preprocessor directives has achieved this goal.

6.6.1 Expected Improvements in our Framework

The initial framework implementation relies on the usage of abstract classes to provide variability for the join algorithms, page replacement strategies, cursors and buffer manager. Because of the extensive usage of abstract classes in our small framework and before-mentioned disadvantages of them, we expect deficits between the initial framework and the preprocessor work in the footprint of the application and also in its performance. In the following, we present reasons for these two assumptions to clarify why it is important to show that these assumptions hold.

Smaller Footprint

We compiled the program using the gcc with *O3* optimization. This enables fast code and the compiler optimizes code by inlining. We expect that the compiler is able to inline some of our abstract classes, such as the page replacement strategy CLOCK to gain performance, because it is the only class inheriting from the abstract class `PageReplacement`. However, many derived classes, such as different buffer managers, will not be able to be inlined easily. As a consequence, unnecessary code providing variability will still be included in the resulting program.

Since preprocessor directives filter out unused code, before the program is compiled, we state that the resulting program will have a smaller footprint. For instance, different implementations of the buffer manager do not have to be available, reducing the compiled code. This may also influence the performance of our framework.

Improved Performance of our Framework

As we have stated above, we expect that the resulting program will have a smaller footprint considering the preprocessor approach. Since programming instructions have to be loaded into the instruction cache to be executed, the executed code should be as small as possible. As a consequence, a reduced footprint should improve the performance.

Another aspect influencing the performance of database operations are caused by the usage of virtual functions in our abstract classes. Using virtual functions, we implicitly access the virtual function table to locate code to be executed. Thus, we argue that calling a high amount of virtual functions causes performance deficits to the preprocessor approach.

6.6.2 Comparison of Both Implementations

For comparing both implementations considering the footprint and performance, we compile each implementation of the framework with exactly the same configuration and measure the two properties. For determining the footprint of our programs, we

simply take the size of the compiled source as a first examination of the footprint of our program.

Another interesting indicator would be the memory consumption of our framework. However, we cannot make clear decisions about memory consumption, because the memory consumption of processed data and intermediate data structures is several cardinalities higher than the savings from the preprocessor directive usage. Thus, measuring the memory consumption does not lead to significant results.

Footprint of Applications

To measure the footprint of our framework, we compiled the four configurations of our framework with the same workload and depict their sizes in Table 6.5. Obviously, we are able to decrease the footprint of the variants by about 6 to 12% using preprocessor directives. When restructuring more complex programs which usually use a high amount of abstract classes and virtual functions, we suppose that footprint savings are even higher.

Footprint in KB	Object-Oriented Approach	Preprocessor Approach
Storage on HDD/SSD	258,6	233 (9,89% less)
SSD as Extended Buffer	262,7	246,4 (6,2% less)
Storage in RAM	253,8	222,7 (12,25% less)

Table 6.5: Footprint of the four configurations of our framework.

Performance Comparison

The most significant results are achieved between the two approaches of the extended buffer pool variant, because it uses two page replacement strategies. Considering the performance of both implementations, we encounter performance improvements of 1 to 33%, as we depict in Figure 6.4. Here, the best performance gain is achieved when the available RAM can hold 40% of the data, because here, functional calls of both page replacement strategies are higher than if one of the buffer managers holds much more pages than the other.

However, we notice that joining small tables will benefit less of the preprocessor approach than joins on bigger tables. Joining, for instance, the `nation` and `region` table may also have a slightly worse performance than the object-oriented approach, which is explainable by normal deviation of our results. Furthermore, since page replacements represent a performance overhead in the object-oriented approach, joining small tables is less beneficial.

We argue that increasing table sizes would favor the preprocessor approach and also more complex implementations that normally use numerous abstract classes will benefit even more from a restructuring using preprocessor directives.

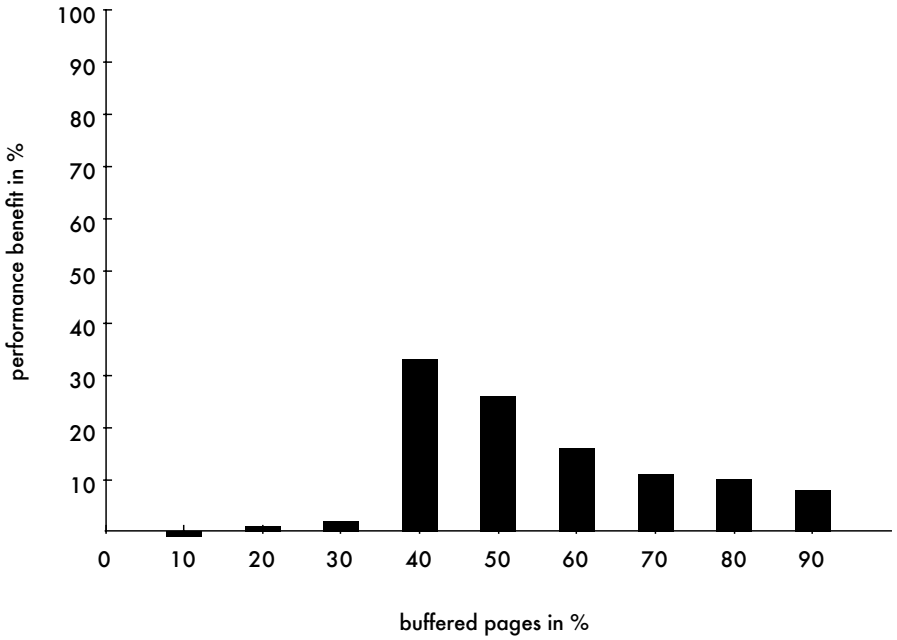


Figure 6.4: Performance benefit for varying RAM sizes for block-nested-loops when joining `supplier` and `lineitem` table using preprocessor implementation of the configuration with SSD as extended buffer pool.

6.7 Threats to Validity

The results that we present in this chapter rely on the proper functionality of our implementation and assumptions that we described in Chapter 5. To justify that we achieved valid results, we discuss important points that may influence measured response times as internal validity and state respective countermeasures to minimize these effects. Furthermore, we state to which extent our results are generalizable to database operations on database systems in the section about external validity.

6.7.1 Internal Validity

An analysis of the internal validity shows to what extent our gathered data is correct and is not biased by other impact factors. To this end, we show how we minimize systematic errors in our implementation.

Implementation of our Framework

For our framework, we have made some assumptions that limit the functionality of our framework regarding the traditional database design. However, we argue that our implementation delivers valid performance results of database operations. Furthermore, we state that our limitations to the five-level-schema architecture do not influence the performance of database operations significantly, as we describe in Section 5.2.2.

Unbiased Implementation of Algorithms

Since we compare the performance of different join algorithms, it is important that we implement each strategy on the same optimization level. With this, we guarantee that our comparison reflects which strategy is the best for a given use case instead of evaluating the implementation.

Considering used data structures, we rely on the functionality of the C++-standard library. Especially the hash map and sort function from the standard library are used in our implementation. In future work, we have to implement own data structures to provide better performance. Nevertheless, at the current state we use already implemented data structures and hope they provide comparability.

Iterative Approach to Get Valid Results

In our work, we start by analyzing different hardware devices to formulate assumptions. With these assumptions, we evaluate the performance of join algorithms to prove our assumptions. If our results do not meet our expectations, we argue possible impact factors that influence our results with respect to the workload to underline the validity of our framework.

Caching Effects

Usually every operating system is designed for improving the performance of all applications. For this, it holds useful data in RAM and, hence, improves the data access of applications. On the one hand, it improves the performance of our algorithms, because data access is a critical factor for database operations. On the other hand, we want to compare the performance of database operations that cause many data accesses using different access patterns and caching minimizes the impact of data access on the performance. Thus, our results are biased by caching. As we have discussed in Section 6.3.2, there are different possibilities to avoid these caching effects. With this, we are able to compare different storage configurations in an unbiased way.

However, not only the RAM caches used data, also instruction and data caches (in L1-L3 caches) allow access to previously used data and operations. Every time an evaluation of a join algorithm is repeated, some used data may be already cached, which distorts our results. To minimize the impact of these caching effects, our experiments iterate through the join algorithms. With this, different data and instructions are used so that caching data of a single algorithm becomes almost impossible, which gives us unbiased results.

Minimize Random Errors

Every measurement is influenced by random errors. In our experiments, random errors are caused by concurrent processes using the same resources as our framework, such as reading data from disk. To provide robustness against random errors, we repeat every configuration 20 times and compute the mean value excluding outliers. Furthermore, we do reproducibility tests to improve the validity of our results.

6.7.2 External Validity

In the external validity, we discuss to which extent our results are generalizable to database operations on other database management systems. To this end, we cover our workload and our limited functionality with respect to the five-level-schema architecture as factors that influence the external validity.

Workload

Using the typical data warehouse benchmark TPC-H, our performance evaluation holds for most of the data warehouse queries. Since the workload includes several small dimension tables and a huge fact table, our join performance is measured for joining one small and one big table. Consequently, we cannot generalize our results to other use cases where tables have nearly the same amount of tuples each. Furthermore, data distribution and whether tuples are sorted influence the performance as well, making our results less generalizable. As a consequence, we have to extend our evaluation to more use cases to reach a high comprehensibility.

Limited Database Functionality

As we have described in Section 5.2.2, we limit our framework in its functionality compared to the five-level-schema architecture. Important points to be mentioned are:

- In our framework, we assume that intermediate data structures, such as hash tables, can be kept in RAM. In normal database management systems, these structures are stored on pages as well to enable synchronization with the persistent storage. With this, the memory consumption of database operations reduces by causing additional disk accesses. Nevertheless, for our evaluation, we want to compare the same database operation for different storage configurations. As a consequence, our intermediate results must fit RAM, because we also want to evaluate in-memory systems where data structures including all data have to be available in RAM.
- Another influence factor that might correlate with the external validity is that our tables are stored on contiguous pages. With this, we are able to compute the storage location of a tuple by knowing its ID, length, and the first page of the table. In contrast, database management system would store necessary information, such as which page belongs to which table, in an additional data structure. This overhead is omitted, here, since we state that this overhead does not severely influence the performance of database operations. Furthermore, for sequential scans, we can use the information in the header of pages to find the next page without further computational overhead.
- Our framework is also limited in supported data types. At the moment, we support data types of a predefined length and implementations for data types of varying lengths are left open for future work. This simplifies our implementation,

because each row has a fixed size and we are able to easily compute storage locations of tuples. However, for future work, we have to support data types such as `varchar` with varying lengths to emulate common database-management-system functionality.

6.7.3 Conclusion Regarding Threats to Validity

To argue that our results are valid and generalizable, we listed important points that influence validity. Concluding, we have reached a high internal validity resulting of our iterative approach and minimizing random errors, which strengthens our comparison of join algorithms for different storage configurations. Nevertheless, additional tests have to be done where we eliminate the influence of caching, to further verify our results.

Considering the external validity, our workload targets typical data warehouse queries. Furthermore, we have made some assumptions that limit the generalizability of our results. Since implementations of database management systems differ, we have to verify our results under different available database management systems to reach a high external validity.

6.8 Summary of our Performance Evaluation

To summarize, we evaluated the performance of our four join algorithms under the four storage configurations and also examined the influence of our preprocessor implementation. Our evaluation brought the following results.

Traditional Storage Configuration

If tables occupy a small amount of pages and RAM is very limited, the block-nested-loops join is superior to the others. However, with increasing RAM and table sizes, the hash-join is the best join algorithm. Nevertheless, the sort-merge join improves its performance steadily with increasing RAM and we argue that its performance may be superior to the hash join when tables are already sorted.

Replacing HDD by SSD

Unfortunately, we cannot state any results that would indicate a performance advantage when using SSDs instead of HDDs with our evaluation. This is either the result of our workload or caused by caching effects that keep our small data available in RAM. To minimize these influence factors, we present different possibilities in order to get results that reflect the difference between HDD and SSD.

Using SSDs as Extended Buffer Pool

Considering the usage of SSDs as an extension to the buffer pool, we identify performance improvements compared to the traditional approach. However, this performance improvement depends on the amount of available RAM. If about more than 40% of the data fits in RAM, using an SSD to buffer often used pages creates more overhead than it saves computational effort. When comparing the overall performance of our four join algorithms, the hash join is still superior to the other three algorithms, because it causes the least amount of page accesses.

In-Memory Configuration

Our in-memory configuration shows the best performance for every algorithm, because data access does not cause disk accesses. The bottleneck in this system is the memory wall, which means CPU-caches work faster than the access to the RAM. As a consequence, our algorithms have to be adapted to cache-friendly access patterns in future work.

Regarding the best performing join strategy for in-memory databases, we identify the hash join. Nevertheless, we argue that other workload including pre-sorted tables may favor the sort-merge join and, consequently, future work has to address these possibilities.

Implementation of our Framework Using Preprocessor Directives

As already assumed, we achieve improvements when avoiding abstract classes and virtual functions. These improvements are visible in a reduced footprint of the application and a slightly improved performance of our join algorithms. SInce our framework consists only of few used abstract classes, we argue that a more complex system with several abstract classes and numerous subclasses benefit even more from the usage preprocessor directives to avoid the overhead of inheritance.

Conclusion of Join Performance

Our goal was to show that the best join algorithm changes if algorithms are executed on different storage configurations. However, with our limited workload, we could not observe highly differing performance advantages between join algorithms, because the hash join seems to be superior. Nevertheless, we observed several local tendencies that, in combination, may result in a totally different join processing. Especially the sort-merge join has the tendency to perform better than the hash join with increasing RAM. On a workload with sorted data, it may become the best join algorithm.

7. Conclusion

In this work, we concentrated on the performance of database operations under different hardware devices. To conclude our outcome, we summarize main results of our six contributions of this work. These contributions are: (1) deriving assumptions from literature regarding the performance of database operations on specific hardware, (2) presenting impact factors on database operation performance, (3) discretization of impact factors to evaluate in this work, (4) comparing implementation techniques that provide variability, (5) performance evaluation of join algorithms on different storage configurations, (6) performance comparison regarding the two implementation techniques.

Assumptions Derived From Literature

To have an overview on already published results, we reviewed papers of the main conferences and stated their contributions regarding the performance of different database operations considering a specific processing or storage device. From this, we formulate the following assumptions:

- In-memory databases should benefit from column-wise storage and operator-at-a-time processing. To avoid the well-known memory wall, algorithms should be implemented cache-consciously.
- At the moment, multi-core CPUs are said to favor the hash join, but with increasing SIMD capabilities, the sort-merge join should become superior.
- Considering the CPU results, the sort-merge join should perform best on the GPU, because of its high SIMD capabilities. Nevertheless, one study showed the superiority of the hash join on GPUs.
- Relatively new and still unpopular processing devices are FPGAs. One paper proposes a block-nested-loops join, which reveals open issues for future work.

- To bridge the access gap between HDD and RAM, SSDs could be used. However, comprehensive performance evaluations of join algorithms using SSDs as extension of the buffer pool are needed.

As a consequence, we identified a uniform evaluation of algorithms under different hardware devices as an important goal. With this, we would be able to prove our assumptions on a uniform methodology.

Presenting Impact Factors on Database Operation Performance

For a uniform evaluation, we had to analyze requirements and impacts for our evaluation methodology. Arising from our literature review, we identified important impact factors that influence the performance of database operations. The impact factors are grouped into database-specific parameters, hardware parameters, algorithms, and workload and for a well-structured representation of their relation to each other, we created the corresponding feature model.

Discretization of Impact Factors to Evaluate

Regarding the resulting feature model, we computed more than 3.7 million possible configurations without considering continuous values. Since such a high amount of evaluations is not feasible, we contributed a reasonable limitation of our feature model leading to 16 variants to be evaluated. These variants include four different storage configurations, where data is stored on HDD or SSD, data is stored on the HDD and the SSD is used as an extension to the buffer pool, and all data is kept in RAM. Furthermore, we evaluate the four join algorithms nested-loops join, block-nested-loops join, hash join and sort-merge join under different buffer and page sizes.

Comparison of Implementation Techniques for Variability

For a uniform evaluation, we implemented a framework for executing database operations under varying hardware and impact factors. The standard implementation focuses on the use of the object-oriented programming paradigm using abstract classes to provide variability. However, we argue that abstract classes and especially virtual functions may introduce an unwanted performance overhead in our processing and present another possibility to implement variability by using preprocessor directives.

Performance Comparison Regarding Different Storage Configurations

Finally, we evaluated the performance of our join algorithms under the four storage configurations and found the following results:

- On the traditional storage configuration, at limited RAM and tables consuming only few pages, the block-nested-loops join is superior. For increasing table sizes and availability of RAM, the hash join is the best algorithm.

- Considering the SSD as storage device instead of the HDD, we could not observe any performance differences. As a consequence, we give some reasons why our results do not match our expectations.
- For the usage of the SSD to extend the buffer pool, we observe that it is only beneficial when the RAM can hold less than 40 % of the data. Still, even for this configuration, the best join algorithm is the hash join.
- The in-memory configuration shows highly improved performance of join algorithms. However, the hash join is superior to the other join algorithms for every evaluated parameter and workload configuration.

Although, we have not found a highly differing performance between our join algorithms, we argue that there are tendencies that may lead to different performance behaviors. These tendencies have to be evaluated in future work.

Performance Comparison Regarding Implementation Techniques

Using our preprocessor implementation, we observe slight improvements in the footprint of our framework and the performance of our join algorithms. Since our framework uses only a few abstract classes, we argue that more complex systems using several abstract classes and complex inheritance hierarchies benefit even more of the tailoring using preprocessor directives.

8. Future Work

Since our work represents the first step to an evaluation of the performance of different database operations, there are numerous steps that have to be carried out to extend the functionality of our framework and evaluate more impact factors on database operation performance. In the following we present important points to be addressed in future work categorized by their occurrence in our feature model.

Database-Specific Factors

Considering the database-specific factors, we have to do several extensions to the given framework to reach more comprehensiveness. One property that is different to normal database management systems are our supported data types. At the moment, we support data types of fixed length. However, to provide more database management functionality, our framework has to handle data types of varying lengths such as `varchar`.

Furthermore, at the moment, only one page replacement strategy is implemented. Future evaluations should include different page replacement strategies that are proposed in literature to reach more comprehensiveness in our evaluation.

At the moment, our framework works on row-oriented tables. However, considering in-memory configurations, column-oriented storage models are favored more. Consequently, we have to extent our implementation to also support a column-oriented view on our data. With the inclusion of column-oriented tables, the possibility to implement different materialization strategies have to be possible as well to give a conclusive performance evaluation.

Considering our processing models, we have to extend our system to also support operator-at-a-time processing. Especially caches should benefit from this processing, because they incur less instruction cache misses.

Hardware Parameters

There are still open questions arising from our evaluation considering the usage of SSDs replacing the HDD. Unfortunately, in our evaluation, we did not observe differences between the usage of the SSD or the HDD as storage device. This phenomenon could be explained because our algorithms cause too less random accesses, to many writes, suboptimal page sizes are chosen, or caching effects benefit the HDD. To eliminate these impacts, we have to repeat our evaluation for bigger tables or under exhausted RAM capacity.

Considering our processing devices, our processing is done single-threaded at the moment. For our database algorithms, it could be beneficial to partition the data and perform the database operations under several threads each having its own partition of data. Hence, parallelization of database operations is left open for future work.

Furthermore, our evaluations should be repeated on an FPGA and GPU. Especially the GPU with its SIMD capabilities may be beneficial for the sort-merge join and, hence, is worth an evaluation. Additionally, since the FPGA is still pretty uncommon for database operations, it yields a big potential for future work. Currently, only block-nested-loops joins were executed on the FPGA and further implementations and evaluations are required to make reasonable statements about advantageous properties of FPGAs for database co-processing.

Database Operation Implementations

At the moment, our performance evaluations are limited to the four join algorithms nested-loops, block-nested-loops, hash and sort-merge join. For a comprehensive evaluation, different implementations of the sort and hash functions have to be evaluated. Since the implementation of the best sort function is related to the best sort operation, further evaluations for remaining database operations have to be done. This includes sorting, aggregation and selection.

Considering complex query plans, we should evaluate the processing of pipelined database operations. For instance, a selection could be executed on one table and the result is joined with another table. With this possibility, we are able to justify or falsify traditional optimization rules with our framework for the new hardware.

Furthermore, an adaptation of our algorithm design to provide cache-consciousness for in-memory configuration has to be done. With this, we avoid the impact of the memory wall and we are able to examine its impact.

Impact Factors of Workload

Our workload is limited to the standard benchmark for data warehouse applications. As a consequence, our evaluation should be extended to consider different data distributions and table sizes. Furthermore, especially for the sort-merge join, pre-sorted data is an important data property that has to be examined as well as indexed data.

Metrics of Database Operation Performance

Another important point for future work concerns the metrics that are evaluated for the database operations. For our evaluation, we used the response time of algorithms, because we evaluated one database operation. Further metrics could be the throughput when considering several database operations.

An important characteristic of database operations is also the energy consumption of the algorithms. Especially considering green computing, algorithms consuming little amount of energy are needed, which makes an additional evaluation of their energy consumption worth.

A. Appendix

In the appendix, we present our gathered performance measurements. For each value in the following tables, we took 20 repetitions and removed the two fastest and slowest response times. From the resulting 16 values, we computed the average response time in μs and depict it here. To keep the table heads short, we use the following abbreviations: B = buffer size of RAM for HDD or SSD as persistent storage device, or buffer size of SSD when used as extension to the buffer pool, PS = page size, NLJ = nested-loops join, BNLJ = block-nested-loops join, HJ = hash join, SMJ = sort-merge join.

The appendix is structured as follows. We start to introduce the performance results of our object-oriented implementation and present corresponding response times for each of the four storage configurations and continue with our performance results of each of the four storage configuration for the implementation using preprocessor directives.

A.1 Object-Oriented Approach

A.1.1 Traditional Configuration with HDD as Persistent Storage Device

B	NLJ	BNLJ	HJ	SMJ
10	127	129	227	230
20	137	130	231	234
30	134	130	229	235
40	136	48	48	150
50	182	48	47	84
60	165	48	48	88
70	73	44	35	37
80	68	43	35	40
90	68	43	34	37

`nation` join `region` at 4KB page size

B	NLJ	BNLJ	HJ	SMJ
10	104	92	252	255
20	108	91	263	256
30	106	88	259	251
40	106	90	258	257
50	168	91	258	255
60	55	41	35	43
70	57	40	37	46
80	54	41	32	43
90	55	42	33	39

`nation` join `region` at 8KB page size

B	NLJ	BNLJ	HJ	SMJ
10	131	111	364	362
20	134	111	362	360
30	140	111	358	358
40	136	110	362	366
50	138	110	356	357
60	69	47	41	42
70	82	45	40	43
80	77	44	39	44
90	341	41	38	47

`nation` join `region` at 16KB page size

B	NLJ	BNLJ	HJ	SMJ
10	1123	1513	810	831
20	1279	671	267	500
30	1211	618	114	422
40	1122	615	113	423
50	1135	618	104	336
60	1128	607	103	257
70	1245	602	102	258
80	540	606	104	143
90	551	610	92	112

`nation` join `supplier` at 4KB page size

B	NLJ	BNLJ	HJ	SMJ
10	964	1196	903	915
20	979	1212	905	923
30	977	585	87	430
40	990	581	87	427
50	1003	575	90	431
60	994	586	85	321
70	1071	583	86	320
80	552	563	83	110
90	551	570	86	114

`nation` join `supplier` at 8KB page size

B	NLJ	BNLJ	HJ	SMJ
10	1025	1201	1340	1337
20	1014	1200	1339	1342
30	1016	1197	1337	1335
40	1050	565	90	378
50	1058	557	96	385
60	1084	559	91	379
70	539	539	85	109
80	534	539	85	112
90	529	543	84	112

`nation` join `supplier` at 16KB page size

B	NLJ	BNLJ	HJ	SMJ
10	16383	8950	1313	5941
20	16235	8712	1326	5818
30	16311	8733	1303	5749
40	16282	8611	1264	5514
50	16150	8551	1176	4502
60	16352	8494	1150	4196
70	16101	8423	1131	3360
80	16151	8367	1120	2909
90	16179	8341	1104	2139

`nation` join `customer` at 4KB page size

B	NLJ	BNLJ	HJ	SMJ
10	12823	8083	1097	5620
20	12851	8151	1086	5354
30	12846	8165	1047	5257
40	13025	8102	1018	5231
50	12981	8173	1001	5110
60	12967	8039	986	3924
70	12843	8048	972	3695
80	13081	7979	964	2951
90	13073	7900	955	2114

`nation` join `customer` at 8KB page size

B	NLJ	BNLJ	HJ	SMJ
10	11631	7937	1077	5719
20	11780	8001	1069	5305
30	11821	7960	1063	5157
40	11775	7979	982	5005
50	11841	7992	969	4941
60	11700	7926	950	4826
70	11909	7942	945	4742
80	12115	7892	938	3938
90	12027	7800	936	2627

`nation` join `customer` at 16KB page size

B	NLJ	BNLJ	HJ	SMJ
10	2265501	1353062	44588	292937
20	2310508	1349168	44416	294683
30	2364565	1334618	43765	272841
40	2371004	1321191	43192	244117
50	2371629	1338200	42497	219763
60	2374726	1309719	41818	190953
70	2361495	1297175	41145	158623
80	2368053	1290375	40492	129700
90	2380012	1287772	39880	101289

`supplier` join `lineitem` at 4KB page size

B	NLJ	BNLJ	HJ	SMJ
10	1895533	1339623	40256	296341
20	1972448	1330637	40380	324242
30	2011974	1321228	39980	336060
40	2008404	1307344	39640	268267
50	2013284	1293293	39118	265882
60	2020866	1282381	38644	223303
70	2021924	1284416	38218	175057
80	2016093	1267810	37706	144859
90	2015204	1256938	37239	105351

`supplier` join `lineitem` at 8KB page size

B	NLJ	BNLJ	HJ	SMJ
10	1783067	1334487	38496	369004
20	1824250	1326472	38764	408206
30	1904210	1308613	38577	431933
40	1912141	1303270	38471	439028
50	1913167	1290362	37906	434277
60	1905996	1277029	37518	301582
70	1897547	1264311	37131	291254
80	1897070	1255253	36727	193459
90	1879094	1243861	36339	132200

`supplier` join `lineitem` at 16KB page size

A.1.2 Configuration with SSD as Persistent Storage Device

B	NLJ	BNLJ	HJ	SMJ
10	126	126	226	228
20	135	127	230	236
30	131	129	231	234
40	137	48	49	158
50	232	51	45	88
60	180	50	46	90
70	75	43	34	36
80	68	43	35	36
90	67	43	35	40

`nation` join `region` at 4KB page size

B	NLJ	BNLJ	HJ	SMJ
10	99	87	254	257
20	119	92	260	255
30	104	85	260	256
40	106	85	264	254
50	176	91	256	253
60	56	42	33	42
70	56	40	30	37
80	58	41	31	40
90	59	40	30	42

`nation` join `region` at 8KB page size

B	NLJ	BNLJ	HJ	SMJ
10	141	107	355	359
20	134	110	348	354
30	136	109	352	354
40	136	110	357	359
50	134	107	352	354
60	67	47	38	41
70	71	46	37	40
80	110	46	37	41
90	248	42	39	44

`nation` join `region` at 16KB page size

B	NLJ	BNLJ	HJ	SMJ
10	1119	1532	825	838
20	1291	668	269	508
30	1214	615	113	425
40	1128	615	111	424
50	1146	619	100	335
60	1128	606	99	256
70	1271	599	99	256
80	534	598	101	142
90	553	607	88	105

`nation` join `supplier` at 4KB page size

B	NLJ	BNLJ	HJ	SMJ
10	973	1221	911	942
20	981	1211	911	933
30	993	582	86	434
40	1001	575	86	436
50	1009	577	86	431
60	996	575	85	320
70	1072	566	89	326
80	541	559	81	103
90	535	561	82	103

`nation` join `supplier` at 8KB page size

B	NLJ	BNLJ	HJ	SMJ
10	1024	1204	1329	1340
20	1019	1214	1337	1342
30	1018	1207	1323	1320
40	1047	565	89	375
50	1062	565	88	373
60	1080	558	88	370
70	536	545	84	107
80	534	554	78	102
90	536	548	81	105

`nation` join `supplier` at 16KB page size

B	NLJ	BNLJ	HJ	SMJ
10	16384	8909	1319	6022
20	16271	8712	1314	5888
30	16289	8695	1306	5828
40	16279	8566	1266	5580
50	16208	8528	1172	4515
60	16364	8445	1144	4181
70	16152	8361	1127	3376
80	16170	8310	1119	2905
90	16209	8287	1095	2165

`nation` join `customer` at 4KB page size

B	NLJ	BNLJ	HJ	SMJ
10	13025	8141	1133	5689
20	12946	8242	1114	5405
30	12977	8195	1051	5339
40	13149	8196	1031	5270
50	13059	8118	1012	5177
60	13085	8078	992	4023
70	12978	8019	980	3797
80	13179	8031	967	3028
90	13255	7959	962	2163

`nation` join `customer` at 8KB page size

B	NLJ	BNLJ	HJ	SMJ
10	11859	8242	1101	5755
20	11814	8038	1079	5352
30	11802	8064	1068	5174
40	11858	8079	984	5088
50	11868	7989	972	5012
60	11774	7972	951	4889
70	11970	8002	953	4848
80	12131	7920	940	4042
90	12184	7879	938	2703

`nation` join `customer` at 16KB page size

B	NLJ	BNLJ	HJ	SMJ
10	2277025	1364455	44693	295541
20	2315991	1358356	44456	296500
30	2375553	1345343	43907	273884
40	2382992	1330930	43194	245252
50	2380642	1346413	42656	220182
60	2384080	1320177	41939	190628
70	2370290	1309288	41312	159219
80	2362712	1301185	40663	130143
90	2392554	1299697	40036	101400

`supplier` join `lineitem`
at 4KB page size

B	NLJ	BNLJ	HJ	SMJ
10	1895456	1347056	40387	298488
20	1958584	1335473	40476	326918
30	2016777	1315447	40124	338738
40	2019597	1309133	39702	270023
50	2018440	1293575	39219	267409
60	2016907	1280511	38765	225214
70	2028193	1286527	38410	176271
80	2016353	1268319	37793	145901
90	2018409	1259188	37359	105866

`supplier` join `lineitem`
at 8KB page size

B	NLJ	BNLJ	HJ	SMJ
10	1785801	1334532	38713	366711
20	1834314	1332196	38946	405961
30	1896835	1314373	38813	428473
40	1907300	1302115	38566	436680
50	1913858	1293110	38024	430821
60	1897605	1279280	37747	300376
70	1896830	1263044	37232	290464
80	1873353	1256379	36931	193116
90	1869961	1247109	36497	131649

`supplier` join `lineitem`
at 16KB page size

A.1.3 Configuration with SSD as Extension to the Buffer Pool

B	NLJ	BNLJ	HJ	SMJ
10	166	134	99	111
20	174	133	96	108
30	176	133	95	108
40	137	137	29	36
50	137	134	28	37
60	136	132	29	37
70	135	133	28	36
80	136	131	27	37
90	137	124	74	77
100	137	130	80	80

`nation` join `region` at 4KB page size
and 10% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	178	133	88	87
20	176	131	102	101
30	175	132	104	101
40	137	126	26	33
50	139	128	27	36
60	142	132	80	78
70	143	133	81	80
80	144	135	81	80
90	144	135	80	80
100	145	135	81	80

`nation` join `region` at 4KB page size
and 20% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	183	135	83	85
20	178	133	82	84
30	180	134	83	85
40	141	132	81	78
50	144	138	81	79
60	147	136	81	79
70	148	134	82	80
80	150	134	81	80
90	150	134	81	80
100	150	134	80	80

`nation` join `region` at 4KB page size and 30% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	138	57	35	48
20	139	57	37	49
30	140	52	36	50
40	143	55	36	49
50	144	56	36	53
60	148	63	36	54
70	147	66	36	51
80	147	65	36	49
90	147	66	36	54
100	150	68	36	53

`nation` join `region` at 4KB page size and 40% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	137	62	36	48
20	140	60	35	48
30	142	61	36	49
40	146	67	36	48
50	149	70	36	45
60	147	74	36	48
70	148	72	37	45
80	149	73	36	46
90	147	74	36	46
100	149	76	36	43

`nation` join `region` at 4KB page size and 50% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	137	63	37	46
20	140	68	36	46
30	143	67	36	45
40	145	77	37	41
50	147	75	37	41
60	146	77	37	42
70	145	77	39	41
80	147	76	39	42
90	146	73	40	41
100	146	76	39	41

`nation` join `region` at 4KB page size and 60% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	61	60	32	35
20	64	59	33	33
30	68	65	32	27
40	69	63	31	29
50	68	64	34	29
60	70	65	33	29
70	68	64	33	29
80	68	66	34	30
90	69	64	33	29
100	69	67	34	31

`nation` join `region` at 4KB page size and 70% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	61	60	35	30
20	66	60	35	33
30	66	59	33	31
40	64	60	34	29
50	65	62	30	35
60	67	59	32	31
70	65	59	30	35
80	66	60	35	28
90	66	60	35	32
100	66	59	34	32

`nation` join `region` at 4KB page size and 80% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	63	61	34	26
20	61	58	37	28
30	64	62	34	33
40	64	59	32	30
50	62	60	36	33
60	61	59	35	29
70	62	60	37	30
80	62	60	34	33
90	61	59	35	36
100	61	61	37	28

`nation` join `region` at 4KB page size and 90% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	120	98	129	121
20	117	97	127	119
30	117	94	130	117
40	119	91	131	122
50	121	99	132	123
60	123	100	133	125
70	124	100	135	128
80	125	100	136	127
90	126	99	135	128
100	126	100	135	129

`nation` join `region` at 8KB page size and 10% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	119	99	130	124
20	119	99	131	121
30	120	100	134	125
40	123	100	133	124
50	123	100	133	123
60	126	100	135	126
70	127	100	134	128
80	129	99	135	129
90	129	100	135	128
100	129	98	132	125

`nation` join `region` at 8KB page size and 20% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	124	100	132	120
20	124	100	131	122
30	125	100	133	123
40	127	100	135	123
50	130	99	133	125
60	133	100	136	130
70	134	100	134	122
80	135	100	133	124
90	136	99	132	122
100	135	100	133	124

`nation` join `region` at 8KB page size and 30% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	128	101	137	129
20	127	100	134	126
30	129	99	135	127
40	132	100	135	126
50	136	101	135	129
60	137	101	135	129
70	138	101	133	125
80	138	101	134	127
90	137	103	135	127
100	137	102	136	129

`nation` join `region` at 8KB page size and 40% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	128	102	130	124
20	126	105	132	130
30	129	102	131	126
40	132	103	135	129
50	134	102	134	129
60	135	101	134	130
70	135	105	135	128
80	136	104	135	128
90	136	105	133	130
100	137	103	131	130

`nation` join `region` at 8KB page size and 50% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	51	49	32	35
20	53	49	30	40
30	59	50	32	38
40	61	50	33	40
50	63	50	33	39
60	65	49	34	39
70	66	50	33	39
80	64	50	32	40
90	63	52	33	38
100	64	51	33	37

`nation` join `region` at 8KB page size and 60% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	55	59	32	42
20	59	56	35	40
30	64	50	33	40
40	62	50	33	40
50	63	52	31	39
60	63	51	33	39
70	64	52	34	40
80	62	51	33	39
90	64	51	34	40
100	63	52	35	38

`nation` join `region` at 8KB page size and 70% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	58	71	36	38
20	58	58	32	40
30	60	59	36	39
40	59	57	32	40
50	59	58	32	43
60	60	59	34	40
70	60	61	36	38
80	61	60	32	39
90	60	62	33	39
100	62	61	32	42

`nation` join `region` at 8KB page size and 80% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	57	67	33	40
20	59	67	28	39
30	59	64	31	40
40	64	66	29	39
50	58	63	31	40
60	59	67	30	39
70	60	66	31	41
80	60	66	31	40
90	61	65	29	41
100	59	67	29	39

`nation` join `region` at 8KB page size and 90% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	160	114	135	132
20	166	113	132	131
30	170	115	131	128
40	173	118	134	131
50	178	121	136	135
60	181	124	139	136
70	182	123	137	137
80	185	124	140	138
90	182	125	139	138
100	178	121	136	134

`nation` join `region` at 16KB page size and 10% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	190	123	140	135
20	187	130	141	139
30	185	137	143	140
40	186	137	141	140
50	186	139	142	138
60	185	138	139	136
70	182	137	139	135
80	182	140	141	137
90	185	146	142	140
100	185	144	138	138

`nation` join `region` at 16KB page size and 20% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	185	153	143	141
20	185	150	143	140
30	185	154	143	141
40	186	153	149	142
50	180	143	141	137
60	182	148	143	138
70	180	142	139	137
80	183	148	146	141
90	182	144	141	138
100	178	142	141	137

`nation` join `region` at 16KB page size and 30% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	187	153	158	146
20	186	150	156	149
30	192	153	163	154
40	183	148	148	144
50	181	146	146	144
60	188	151	153	147
70	181	144	145	142
80	187	148	153	149
90	183	147	150	150
100	181	145	146	143

`nation` join `region` at 16KB page size and 40% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	716	158	166	171
20	190	150	155	157
30	192	154	160	161
40	189	150	154	154
50	182	147	147	147
60	179	146	146	146
70	180	143	146	146
80	183	151	150	147
90	185	148	153	152
100	184	148	150	144

`nation` join `region` at 16KB page size and 50% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	345	83	45	50
20	94	86	46	53
30	98	87	47	55
40	97	89	48	51
50	98	88	47	50
60	98	88	47	50
70	98	88	47	51
80	98	87	46	49
90	96	86	44	48
100	98	87	47	51

`nation` join `region` at 16KB page size and 60% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	156	80	46	48
20	98	86	48	51
30	98	87	47	50
40	99	89	49	52
50	96	87	47	50
60	99	88	48	52
70	98	89	48	53
80	98	88	47	52
90	97	88	46	49
100	97	88	49	54

`nation` join `region` at 16KB page size and 70% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	97	87	48	51
20	98	89	47	51
30	97	87	48	51
40	98	89	49	51
50	96	88	48	109
60	98	89	158	45
70	97	154	43	52
80	116	175	43	51
90	192	83	49	50
100	149	86	49	52

`nation` join `region` at 16KB page size and 80% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	96	87	48	51
20	95	87	48	51
30	95	87	48	50
40	96	87	47	50
50	96	87	48	52
60	95	87	48	51
70	95	86	48	51
80	96	87	48	51
90	96	87	48	51
100	96	87	48	51

`nation` join `region` at 16KB page size and 90% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1506	1551	665	1677
20	1522	1546	429	1633
30	1536	1546	462	1623
40	1525	1548	468	1642
50	1510	1539	335	1390
60	1135	1524	298	1428
70	1126	1535	300	1438
80	1117	1524	334	1445
90	1131	1520	429	1650
100	1125	1528	453	1721

`nation` join `supplier` at 4KB page size and 10% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1523	669	208	433
20	1508	662	176	426
30	1518	658	192	434
40	1517	659	193	437
50	1130	658	146	364
60	1148	649	205	358
70	1140	643	206	360
80	1131	651	208	361
90	1146	646	206	361
100	1139	647	206	360

`nation` join `supplier` at 4KB page size and 20% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1526	608	155	380
20	1541	602	152	378
30	1145	601	150	298
40	1129	607	150	300
50	1140	602	148	297
60	1150	599	147	298
70	1137	602	148	299
80	1134	603	148	299
90	1126	607	150	300
100	1140	597	149	303

`nation` join `supplier` at 4KB page size and 30% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1533	609	156	384
20	1533	605	152	381
30	1142	601	150	300
40	1127	606	150	303
50	1128	604	148	297
60	1127	608	148	300
70	1129	606	148	300
80	1129	606	149	299
90	1127	607	148	300
100	1127	610	148	300

`nation` join `supplier` at 4KB page size and 40% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1533	594	138	372
20	1133	593	136	289
30	1126	592	132	292
40	1136	591	133	293
50	1138	591	133	295
60	1137	590	133	295
70	1137	597	133	295
80	1144	594	134	295
90	1146	593	133	295
100	1140	598	134	295

`nation` join `supplier` at 4KB page size and 50% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1133	588	132	287
20	1125	586	130	287
30	1127	585	130	287
40	1133	584	132	289
50	1134	590	131	290
60	1138	588	132	291
70	1137	589	131	290
80	1140	584	131	290
90	1129	590	131	290
100	1129	586	132	290

`nation` join `supplier` at 4KB page size and 60% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1118	599	143	291
20	1128	589	133	289
30	1130	590	133	291
40	1125	598	135	289
50	1125	584	134	291
60	1130	591	133	290
70	1130	594	133	293
80	1128	589	132	290
90	1125	592	133	290
100	1116	594	139	291

`nation` join `supplier` at 4KB page size and 70% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	590	589	136	165
20	591	592	144	176
30	591	584	142	175
40	597	594	136	168
50	593	596	133	166
60	597	593	138	168
70	599	595	138	171
80	591	584	143	173
90	589	595	145	173
100	591	595	142	170

`nation` join `supplier` at 4KB page size and 80% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	560	583	129	163
20	557	591	132	163
30	562	583	131	162
40	560	584	136	162
50	560	584	131	164
60	556	588	138	164
70	562	585	133	164
80	560	584	132	165
90	565	581	132	161
100	554	582	133	163

`nation` join `supplier` at 4KB page size and 90% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1313	1231	545	3123
20	1293	1230	558	3483
30	1314	1215	556	3156
40	1293	1210	540	3114
50	1286	1208	544	3114
60	982	1208	540	3054
70	981	1202	540	3057
80	978	1218	547	3063
90	983	1205	540	3039
100	980	1211	543	3073

`nation` join `supplier` at 8KB page size and 10% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1284	1214	532	3404
20	1274	1217	534	3408
30	1288	1211	543	3117
40	1303	1206	546	3117
50	1294	1210	545	3128
60	983	1207	542	3046
70	972	1205	543	3042
80	969	1211	543	3047
90	979	1201	545	3035
100	981	1201	544	3038

`nation` join `supplier` at 8KB page size and 20% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1302	560	167	539
20	1297	563	168	539
30	994	570	157	420
40	991	571	159	423
50	989	565	159	423
60	984	563	158	426
70	978	563	160	427
80	985	564	159	426
90	992	560	158	425
100	986	560	158	425

`nation` join `supplier` at 8KB page size and 30% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1307	568	168	543
20	1300	565	168	544
30	981	563	157	419
40	979	563	158	421
50	982	565	158	425
60	980	564	159	425
70	985	566	161	426
80	987	559	160	424
90	987	566	159	425
100	980	570	159	428

`nation` join `supplier` at 8KB page size and 40% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1313	571	170	543
20	1325	571	169	552
30	988	565	164	422
40	983	561	163	424
50	983	565	160	429
60	983	565	161	429
70	988	563	160	427
80	974	568	159	425
90	995	564	161	429
100	991	566	159	426

`nation` join `supplier` at 8KB page size and 50% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1001	566	165	374
20	992	566	169	375
30	993	563	168	376
40	993	571	171	380
50	988	568	171	376
60	983	567	173	379
70	983	566	169	378
80	992	573	167	376
90	1003	566	167	378
100	1005	566	167	379

`nation` join `supplier` at 8KB page size and 60% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	988	564	168	371
20	989	570	168	378
30	995	564	167	379
40	1001	564	170	380
50	989	571	165	377
60	995	568	168	378
70	1002	568	166	377
80	991	567	169	376
90	997	565	172	379
100	999	567	168	378

`nation` join `supplier` at 8KB page size and 70% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	553	563	163	227
20	553	567	169	238
30	545	555	169	230
40	546	558	166	232
50	551	565	163	228
60	555	557	163	228
70	552	560	162	229
80	548	561	167	234
90	554	564	166	234
100	557	556	161	228

`nation` join `supplier` at 8KB page size and 80% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	550	560	163	230
20	562	559	160	228
30	550	558	165	230
40	549	562	164	227
50	547	563	161	226
60	545	557	167	229
70	553	558	163	229
80	548	556	165	230
90	556	557	164	230
100	551	558	168	235

`nation` join `supplier` at 8KB page size and 90% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1423	1190	1184	6680
20	1430	1186	1187	6919
30	1449	1192	1182	6941
40	1042	1209	1237	10486
50	1049	1208	1242	10504
60	1049	1213	1263	10502
70	1054	1216	1280	10548
80	1056	1216	1285	10531
90	1063	1224	1304	10674
100	1061	1217	1290	10579

`nation` join `supplier` at 16KB page size and 10% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1494	1230	1246	7055
20	1492	1225	1246	7056
30	1499	1224	1246	7031
40	1067	1222	1302	10707
50	1071	1221	1304	10745
60	1078	1219	1299	10709
70	1073	1226	1304	10789
80	1082	1225	1309	10812
90	1070	1231	1309	10794
100	1059	1229	1305	10805

`nation` join `supplier` at 16KB page size and 20% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1500	1226	1251	7154
20	1492	1230	1246	7158
30	1498	1228	1251	7157
40	1084	1227	1308	10824
50	1061	1225	1312	10888
60	1080	1228	1309	10870
70	1074	1227	1315	10893
80	1081	1233	1324	11004
90	1069	1232	1309	10906
100	1062	1226	1311	10855

`nation` join `supplier` at 16KB page size and 30% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1079	570	217	460
20	1084	573	213	463
30	1088	576	214	463
40	1084	573	210	462
50	1077	572	214	462
60	1083	579	211	462
70	1080	579	212	463
80	1082	572	214	463
90	1084	571	212	463
100	1083	575	212	462

`nation` join `supplier` at 16KB page size and 40% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1075	572	214	457
20	1082	572	212	461
30	1080	582	212	462
40	1087	571	212	462
50	1076	579	214	464
60	1078	576	212	463
70	1089	579	212	463
80	1086	572	212	464
90	1089	574	211	463
100	1080	575	212	463

`nation` join `supplier` at 16KB page size and 50% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1171	574	212	462
20	1121	576	212	461
30	1132	573	211	462
40	1137	574	212	464
50	1134	583	214	464
60	1135	578	211	463
70	1120	574	218	463
80	1114	579	211	462
90	1128	573	212	462
100	1147	578	211	463

`nation` join `supplier` at 16KB page size and 60% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	645	568	206	295
20	617	559	203	301
30	621	560	204	300
40	631	567	205	299
50	624	565	211	303
60	631	562	208	304
70	620	567	203	298
80	612	565	209	301
90	616	568	207	302
100	633	563	205	300

`nation` join `supplier` at 16KB page size and 70% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	661	565	206	298
20	760	563	207	300
30	676	572	204	298
40	681	568	204	297
50	624	576	203	297
60	637	568	205	297
70	619	572	203	299
80	637	567	212	303
90	628	566	214	310
100	631	569	206	297

`nation` join `supplier` at 16KB page size and 80% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	788	573	202	297
20	785	567	208	302
30	773	566	202	297
40	788	563	212	298
50	785	565	205	301
60	784	565	204	297
70	760	571	203	295
80	785	564	205	298
90	789	570	203	298
100	773	567	204	296

`nation` join `supplier` at 16KB page size and 90% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	21853	8716	1264	3984
20	21710	8787	1288	4187
30	21702	8795	1262	4157
40	21773	8767	1254	4175
50	21790	8733	1227	4165
60	21676	8785	1201	4101
70	21647	8756	1173	4115
80	21728	8726	1161	4128
90	16222	8641	1320	3529
100	16255	8660	1320	3596

`nation` join `customer` at 4KB page size and 10% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	22212	9166	1390	4795
20	22185	9136	1258	4229
30	22203	9133	1243	4234
40	22270	9090	1219	4212
50	22348	9081	1196	4207
60	22308	9071	1312	5012
70	22252	9052	1292	5001
80	16673	8762	1379	3808
90	16651	8774	1372	3815
100	16647	8780	1386	3799

`nation` join `customer` at 4KB page size and 20% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	22627	9340	1403	5124
20	22607	9303	1380	5104
30	22712	9324	1385	5112
40	22701	9313	1366	5066
50	22753	9280	1339	5043
60	22670	9281	1302	5019
70	17100	9096	1351	3735
80	17080	9124	1352	3728
90	17091	9160	1361	3730
100	17059	9126	1346	3732

`nation` join `customer` at 4KB page size and 30% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	22996	9535	1373	5084
20	22944	9491	1343	5128
30	22944	9566	1329	5087
40	22934	9548	1305	5086
50	22930	9522	1283	5030
60	17411	9292	1336	3759
70	17381	9309	1345	3752
80	17353	9307	1342	3738
90	17374	9345	1345	3749
100	17381	9342	1343	3747

`nation` join `customer` at 4KB page size and 40% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	23338	9757	1338	5098
20	23310	9800	1319	5077
30	23374	9762	1298	5073
40	23439	9728	1273	5072
50	17824	9563	1330	3755
60	17810	9575	1335	3760
70	17852	9593	1316	3754
80	17875	9589	1314	3760
90	17908	9618	1316	3750
100	17875	9613	1315	3743

`nation` join `customer` at 4KB page size and 50% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	23685	9985	1327	5082
20	23667	9974	1288	5056
30	23710	9957	1267	5042
40	18128	9820	1273	3741
50	18186	9832	1270	3723
60	18236	9850	1267	3753
70	18182	9847	1269	3725
80	18134	9876	1268	3721
90	18153	9849	1271	3723
100	18172	9866	1267	3734

`nation` join `customer` at 4KB page size and 60% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	23764	10055	1259	5005
20	23738	10081	1239	4990
30	18240	9910	1244	3661
40	18230	9885	1247	3644
50	18234	9902	1246	3676
60	18226	9924	1245	3680
70	18242	9872	1244	3680
80	18236	9906	1246	3663
90	18243	9938	1248	3635
100	18209	9873	1244	3654

`nation` join `customer` at 4KB page size and 70% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	23872	10133	1224	4936
20	18288	9911	1261	3656
30	18283	9947	1262	3659
40	18351	9961	1265	3691
50	18364	9930	1261	3689
60	18331	9923	1263	3685
70	18341	9951	1261	3682
80	18391	9937	1263	3689
90	18404	9914	1255	3681
100	18484	9953	1248	3692

`nation` join `customer` at 4KB page size and 80% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	18423	9959	1229	3677
20	18433	9940	1232	3666
30	18434	9949	1239	3663
40	18446	9974	1233	3663
50	18458	9973	1225	3673
60	18465	9963	1226	3671
70	18402	9953	1239	3662
80	18422	10002	1245	3652
90	18485	9953	1242	3691
100	18579	9975	1236	3695

`nation` join `customer` at 4KB page size and 90% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	15733	7950	870	1016
20	15614	7920	889	934
30	15793	8010	965	922
40	16163	8306	970	892
50	16214	8303	1011	895
60	16305	8390	1013	884
70	16357	8418	983	872
80	16436	8571	922	858
90	13024	8520	834	968
100	13049	8554	821	974

`nation` join `customer` at 8KB page size and 10% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	16615	8871	830	969
20	16587	8833	826	962
30	16588	8818	839	940
40	16639	8903	863	905
50	16674	8930	858	909
60	16724	9091	774	919
70	16768	8735	891	873
80	13315	8502	1115	909
90	13309	8502	1130	908
100	13287	8476	1113	909

`nation` join `customer` at 8KB page size and 20% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	16873	9158	1346	920
20	16881	9063	1333	913
30	16901	8935	1076	1010
40	16943	8817	1037	981
50	16908	8742	849	1013
60	16965	8741	791	999
70	13514	8508	738	1012
80	13509	8509	733	1022
90	13529	8547	744	1002
100	13539	8546	731	1018

`nation` join `customer` at 8KB page size and 30% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	17208	9430	839	947
20	17208	9368	845	931
30	17163	9212	830	964
40	17225	9135	830	956
50	17212	9086	813	948
60	13852	8839	731	1000
70	13871	8842	734	996
80	13823	8863	740	1001
90	13852	8847	738	999
100	13886	8891	769	1014

`nation` join `customer` at 8KB page size and 40% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	17346	9508	797	944
20	17354	9456	784	934
30	17388	9319	820	951
40	17385	9245	810	948
50	13992	8939	798	1031
60	13989	8977	809	1023
70	13991	8955	801	1036
80	13988	8963	798	1043
90	13978	8972	795	1027
100	13923	8970	791	1019

`nation` join `customer` at 8KB page size and 50% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	17426	9485	825	958
20	17456	9399	815	947
30	17438	9365	802	940
40	14091	9036	831	1014
50	14062	9047	825	1018
60	14064	9070	828	1016
70	14038	9066	829	1017
80	14071	9082	828	1012
90	14079	9072	832	1012
100	14061	9084	843	1011

`nation` join `customer` at 8KB page size and 60% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	17542	9504	813	942
20	17541	9481	809	930
30	14148	9183	810	1005
40	14185	9214	802	1000
50	14190	9184	801	1016
60	14203	9183	809	1012
70	14168	9222	802	1007
80	14201	9205	793	1013
90	14187	9181	810	1015
100	14203	9206	798	1010

`nation` join `customer` at 8KB page size and 70% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	17648	9471	763	1082
20	14227	9278	717	999
30	14257	9209	713	1021
40	14214	9258	708	1012
50	14247	9316	714	993
60	14256	9215	712	1017
70	14226	9257	711	1011
80	14263	9300	715	1003
90	14255	9243	717	1014
100	14237	9253	710	1018

`nation` join `customer` at 8KB page size and 80% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	14406	9251	706	989
20	14373	9283	707	979
30	14448	9224	713	981
40	14423	9280	726	983
50	14443	9239	704	987
60	14419	9264	706	981
70	14446	9242	709	977
80	14426	9260	719	988
90	14423	9254	715	982
100	14375	9234	702	983

`nation` join `customer` at 8KB page size and 90% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	14477	8033	945	2925
20	14471	7940	948	2968
30	14570	7961	948	2994
40	14838	7900	939	2970
50	14816	7896	936	2965
60	14739	7858	916	2938
70	14800	7837	912	2967
80	14778	7869	912	2956
90	11762	7916	904	2159
100	11741	7895	908	2171

`nation` join `customer` at 16KB page size and 10% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	14953	8000	968	3070
20	15006	7987	969	3072
30	15061	7977	957	3090
40	15143	7930	946	3071
50	15166	7973	937	3054
60	15144	7936	927	3048
70	15175	7942	915	3032
80	12171	7950	921	2215
90	12145	7949	925	2215
100	12160	7975	929	2211

`nation` join `customer` at 16KB page size and 20% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	15264	8038	958	3128
20	15287	8032	952	3110
30	15292	8050	950	3107
40	15379	8036	936	3040
50	15374	7996	927	3045
60	15394	7989	914	3047
70	12356	8017	914	2223
80	12364	7996	921	2221
90	12377	8012	919	2218
100	12358	8022	920	2215

`nation` join `customer` at 16KB page size and 30% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	15517	8444	982	3131
20	15488	8425	969	3105
30	15479	8424	959	3086
40	15464	8408	947	3068
50	15501	8405	938	3079
60	12507	8189	985	2417
70	12507	8195	982	2410
80	12480	8189	966	2391
90	12464	8210	992	2397
100	12490	8190	982	2415

`nation` join `customer` at 16KB page size and 40% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	15522	8516	959	3095
20	15545	8465	962	3103
30	15571	8442	947	3084
40	15530	8429	937	3081
50	12589	8201	999	2433
60	12595	8198	983	2438
70	12599	8217	995	2434
80	12602	8214	976	2411
90	12584	8225	1004	2427
100	12618	8216	998	2437

`nation` join `customer` at 16KB page size and 50% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	15589	8492	938	3076
20	15588	8478	932	3063
30	15592	8440	922	3049
40	12646	8250	968	2431
50	12689	8246	998	2435
60	12654	8259	974	2420
70	12657	8271	974	2422
80	12638	8267	983	2417
90	12640	8255	987	2424
100	12656	8270	978	2413

`nation` join `customer` at 16KB page size and 60% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	15738	8497	931	3094
20	15696	8462	930	3094
30	12763	8242	955	2438
40	12763	8244	957	2451
50	12798	8252	961	2420
60	12782	8285	964	2431
70	12784	8276	956	2444
80	12809	8280	968	2440
90	12792	8282	956	2433
100	12787	8307	961	2435

`nation` join `customer` at 16KB page size and 70% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	15810	8530	921	3087
20	12827	8300	970	2450
30	12892	8339	960	2425
40	12877	8333	961	2435
50	12884	8330	967	2443
60	12863	8361	948	2442
70	12903	8333	952	2472
80	12901	8305	969	2507
90	12896	8317	966	2523
100	12883	8337	976	2480

`nation` join `customer` at 16KB page size and 80% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	12882	8343	949	2447
20	12888	8361	936	2429
30	12890	8340	955	2444
40	12916	8370	945	2428
50	12901	8370	947	2437
60	12914	8375	954	2407
70	12900	8384	945	2429
80	12921	8396	980	2419
90	12925	7958	711	2036
100	12891	8380	949	2428

`nation` join `customer` at 16KB page size and 90% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	3947124	2327863	47198	55565
20	3939728	2316784	46730	56568
30	3965173	2309259	46441	56189
40	3975175	2295532	46812	55918
50	3999301	2291101	47338	55277
60	4002348	2291926	47128	54551
70	3991476	2299337	47287	53766
80	3995036	2306901	47225	53143
90	3210327	2309908	46871	52302
100	3212194	2311221	46976	52300

`supplier` join `lineitem` at 4KB page size and 10% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1047156	3204843	63215	66746
20	1037476	3181121	61902	66506
30	857315	3166916	57428	65597
40	837868	3172198	57472	64962
50	800463	3184672	57436	64349
60	809319	3189105	56896	63636
70	760936	3210802	57031	62879
80	4209147	3234025	57264	62059
90	4212787	3257005	58163	62186
100	4207781	3266030	58256	62123

`supplier` join `lineitem` at 4KB page size and 20% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	2060414	4022029	70328	73213
20	2103218	4004577	69602	72622
30	2068751	3995169	68685	71994
40	2067485	3983813	67474	71343
50	1847171	3962941	64817	70562
60	1744237	3955869	63954	69880
70	910787	3960282	63502	69023
80	906636	3973714	63785	69062
90	901507	3978122	64131	69091
100	898457	3997500	64632	69064

`supplier` join `lineitem` at 4KB page size and 30% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	3028059	332874	74756	78254
20	2998465	306459	71806	77595
30	2749997	297739	70227	77031
40	2775206	317401	70311	76428
50	2719333	332091	70373	75816
60	1910043	363409	70331	75190
70	1914253	381458	71235	75065
80	1910651	396003	71321	75218
90	1909374	383188	71221	75118
100	1905295	400071	71276	75120

`supplier` join `lineitem` at 4KB page size and 40% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	4027333	941320	79170	82722
20	3974128	913499	77459	82162
30	3747301	907934	75428	81299
40	3706790	928248	75779	80657
50	2906957	959707	75668	79871
60	2892572	952973	76007	79828
70	2899942	980243	76174	79956
80	2901999	969468	76177	80027
90	2902002	959623	76094	80232
100	2901500	969493	75914	79859

`supplier` join `lineitem` at 4KB page size and 50% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	580118	1403966	79760	85823
20	360284	1418130	79648	84750
30	353626	1411293	79754	84498
40	3885890	1430662	79171	83632
50	3888848	1438785	79267	83571
60	3894890	1443662	79573	83392
70	3890479	1441925	79274	83662
80	3887596	1435434	79280	83765
90	3892900	1424558	78546	83509
100	3890510	1422294	78494	83420

`supplier` join `lineitem`
at 4KB page size
and 60% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1293842	1763616	82022	88151
20	1314414	1738843	81346	86272
30	594984	1713126	80789	86789
40	594783	1774754	80220	86939
50	581281	1762195	80830	86169
60	585031	1716319	80612	85971
70	583905	1756834	79990	86918
80	577810	1774194	80673	85954
90	570722	1706929	80701	85616
100	576971	1745392	79689	86782

`supplier` join `lineitem`
at 4KB page size
and 70% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	2228159	1976390	82526	88935
20	1546246	2020399	82275	87595
30	1561034	2011900	82025	88258
40	1563865	1935955	82134	88512
50	1566228	1995999	82001	88595
60	1561136	2003191	82042	87962
70	1567714	1996645	81693	88481
80	1565859	1943044	82044	88571
90	1593774	2015654	82021	88761
100	1588075	2002773	81885	87794

`supplier` join `lineitem`
at 4KB page size
and 80% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	2497285	2089329	82240	88834
20	2524340	2144535	82255	89536
30	2487274	2105767	81729	89034
40	2478382	2083479	81797	88140
50	2503315	2075605	81923	88894
60	2482361	2104172	81818	88725
70	2481033	2093986	81431	89084
80	2493609	2079373	81611	88607
90	2498980	2055884	81603	88825
100	2502611	2103540	81717	89124

`supplier` join `lineitem`
at 4KB page size
and 90% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	2796421	1963307	40744	42891
20	2784411	1955150	40129	42674
30	2780387	1959844	39619	42290
40	2803554	1958340	39204	41872
50	2819324	1955921	38631	41495
60	2836631	1961639	38053	41067
70	2821745	1962557	37527	40573
80	2792316	1958233	37000	40602
90	2323889	1959511	36472	39297
100	2319634	1958845	36562	39314

`supplier` join `lineitem`
at 8KB page size
and 10% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	3613308	2423086	44492	47189
20	3591598	2418651	44105	47071
30	3379581	2421812	43621	46574
40	3381781	2424652	43264	46260
50	3385504	2416235	42600	45989
60	3367390	2419312	42197	45302
70	3322612	2421580	41782	44626
80	2811581	2419798	41201	44160
90	2811599	2420923	41216	44039
100	2808326	2421361	41264	43928

`supplier` join `lineitem`
at 8KB page size
and 20% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	3978898	2838814	47826	51362
20	3703026	2838054	47370	50503
30	3678083	2837778	46935	50142
40	3381047	2834422	46456	49732
50	3916658	2839235	45929	49222
60	3881057	2835230	45380	48788
70	3352807	2838733	44878	48139
80	3353182	2840934	44857	48174
90	3352808	2837869	44952	48154
100	3354817	2837370	44999	48144

`supplier` join `lineitem`
at 8KB page size
and 30% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	391084	3180749	50278	53382
20	247876	3178467	49964	53113
30	153443	3178085	49477	52612
40	136930	3178216	48928	52250
50	89084	3177112	48343	52225
60	3870702	3175907	47852	51141
70	3961400	3178955	47872	51224
80	4013673	3179424	47961	51193
90	4042657	3174967	48020	51256
100	4082603	3177399	48027	51157

`supplier` join `lineitem`
at 8KB page size
and 40% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	915755	3468845	52131	55689
20	899976	3471444	51702	55401
30	895130	3467605	51337	55013
40	801037	3471534	50668	54516
50	96714	3467440	50181	54047
60	79898	3470833	50262	54105
70	70194	3470094	50294	54063
80	66945	3468487	50283	53956
90	67493	3466575	50327	54060
100	81013	3472455	50405	54068

`supplier` join `lineitem`
at 8KB page size
and 50% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1311625	3699312	54049	58076
20	1077654	3703334	53591	57764
30	1071821	3699143	52984	57227
40	557588	3694607	52555	56527
50	552460	3696266	52631	56624
60	554064	3700979	52613	56654
70	554321	3698348	52698	56637
80	554201	3698264	52621	56618
90	557513	3696470	52673	56626
100	555488	3700279	52662	56699

`supplier` join `lineitem`
at 8KB page size
and 60% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1558475	3878483	54945	59013
20	1510735	3876916	54482	58588
30	1035783	3875719	53931	57918
40	1041115	3875906	53975	57943
50	1036262	3871857	53965	57859
60	1036788	3880579	54077	58062
70	1036198	3877284	54085	58122
80	1033834	3874172	53987	58005
90	1037912	3876635	53939	58109
100	1039148	3997120	55339	59380

`supplier` join `lineitem`
at 8KB page size
and 70% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1990120	3997177	54766	58881
20	1511566	3997157	54846	58976
30	1518159	3995808	54882	58968
40	1511234	3997146	54891	58929
50	1509977	3997421	54944	59127
60	1514041	3994565	54926	58968
70	1512231	3996005	54951	59225
80	1511853	3996442	54873	58981
90	1511509	3994849	55007	58919
100	1519505	4066467	55116	59113

`supplier` join `lineitem`
at 8KB page size
and 80% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1986910	3809838	64070	61880
20	1982560	4062058	55175	58981
30	1992469	4066433	55330	59130
40	1980232	4066733	55428	59252
50	1987452	4063810	55159	59001
60	1984929	4065529	54988	59485
70	1981705	4062939	54930	59170
80	1989456	4062968	54908	59215
90	1987124	4066913	54909	59165
100	1981590	4063511	54849	59067

`supplier` join `lineitem`
at 8KB page size
and 90% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	2384273	1682902	37539	77897
20	2378991	1678729	34649	78149
30	2393778	1680339	34291	78352
40	2408322	1681088	33896	78201
50	2421408	1675809	33523	78041
60	2424862	1681719	33075	77948
70	2428715	1680734	32644	77403
80	2406883	1681399	32169	76063
90	1988071	1680569	31685	57767
100	1990664	1680444	31754	57902

`supplier` join `lineitem`
at 16KB page size
and 10% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	2743751	1903871	36828	81710
20	2751481	1904560	36615	81645
30	2756736	1902067	36139	81803
40	2769105	1906379	35862	81657
50	2762918	1903206	35365	81741
60	2771777	1904528	34827	82518
70	2723392	1902283	34458	79750
80	2246005	1903728	34000	60690
90	2246118	1903918	34117	61063
100	2246468	1904132	34111	61220

`supplier` join `lineitem`
at 16KB page size
and 20% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	3045817	2093579	38217	83821
20	3035597	2099211	37998	84036
30	3064046	2101441	37638	83962
40	3048351	2098382	37222	83919
50	3034886	2097128	36716	83486
60	2997938	2097765	36356	82029
70	2511466	2099016	35909	63445
80	2514991	2099341	35996	63774
90	2512976	2098187	36022	66069
100	2511527	2098660	36052	64255

`supplier` join `lineitem`
at 16KB page size
and 30% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	3282674	2263136	39364	85996
20	3291746	2263393	38988	86182
30	3286796	2261120	38774	85999
40	3281213	2261884	38191	85727
50	3255000	2261096	37828	84875
60	2764758	2260944	37360	66556
70	2765909	2262108	37411	66759
80	2765604	2260324	37446	66959
90	2762170	2260220	37506	67268
100	2766979	2261363	37501	67481

`supplier` join `lineitem`
at 16KB page size
and 40% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	3567694	2414920	40112	87266
20	3570617	2413002	39928	87027
30	3566371	2412811	39373	87007
40	3537557	2411298	38949	85906
50	3072165	2412289	38686	68902
60	3075951	2411473	38692	69171
70	3079351	2409546	38704	69459
80	3075884	2408253	38740	69657
90	3077745	2412236	38819	69843
100	3079611	2411210	38848	69942

`supplier` join `lineitem`
at 16KB page size
and 50% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	3804497	2525278	40655	87736
20	3800262	2524498	40226	87772
30	3771866	2525345	39792	87047
40	3333580	2530332	39491	69747
50	3328451	2526661	39473	70634
60	3335612	2529805	39520	70253
70	3329444	2526204	39524	70405
80	3330607	2528347	39535	70782
90	3328589	2531849	39541	70877
100	3325160	2527687	39632	71398

`supplier` join `lineitem`
at 16KB page size
and 60% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	4041669	2606697	40895	87790
20	4013771	2607116	40533	86316
30	3565621	2603951	40067	69762
40	3568916	2605865	40156	70289
50	3567758	2605778	40232	70739
60	3572538	2606065	40192	70992
70	3570799	2607456	40248	71185
80	3570908	2605746	40168	71176
90	3573800	2605887	40241	71409
100	3571424	2605144	40257	71608

`supplier` join `lineitem`
at 16KB page size
and 70% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	4235352	2665154	40896	86050
20	3809131	2663816	40436	69538
30	3810829	2663150	40530	70201
40	3810525	2664110	40555	70662
50	3810721	2663917	40559	71153
60	3809409	2663338	40508	71450
70	3808352	2663376	40491	71502
80	3804081	2664239	40521	71850
90	3801971	2663529	40456	71841
100	3806953	2663259	40472	71867

`supplier` join `lineitem`
at 16KB page size
and 80% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	4048977	2683098	40643	69054
20	4045603	2689309	40740	70035
30	4049520	2685534	40704	70930
40	4045844	2688547	40738	71367
50	4042612	2683957	40679	71689
60	4043162	2685823	40736	71917
70	4045024	2686627	40782	72078
80	4044980	2684666	40636	71996
90	4050533	2690135	40659	72185
100	4050548	2683231	40696	72266

`supplier` join `lineitem`
at 16KB page size
and 90% available RAM

A.1.4 In-Memory Configuration

PS	NLJ	BNLJ	HJ	SMJ
4KB	40	38	23	32
8KB	39	37	23	29
16KB	38	38	25	33

`nation` join `region`

PS	NLJ	BNLJ	HJ	SMJ
4KB	571	565	74	104
8KB	566	537	77	110
16KB	568	534	77	110

`nation` join `supplier`

PS	NLJ	BNLJ	HJ	SMJ
4KB	8545	8503	830	1227
8KB	8660	8331	821	1215
16KB	8523	8057	820	1206

`nation` join `customer`

PS	NLJ	BNLJ	HJ	SMJ
4KB	1344499	1331276	33914	63643
8KB	1338135	1316403	33512	62925
16KB	1305419	1286363	33428	62008

`supplier` join `lineitem`

A.2 Implementation Using Preprocessor Directives

A.2.1 Traditional Configuration with HDD as Persistent Storage Device

B	NLJ	BNLJ	HJ	SMJ
10	128	133	238	244
20	138	131	243	243
30	133	129	240	243
40	136	47	47	147
50	213	48	47	87
60	172	48	45	86
70	71	43	34	35
80	64	43	34	35
90	63	43	33	36

`nation` join `region` at 4KB page size

B	NLJ	BNLJ	HJ	SMJ
10	105	93	269	268
20	111	93	278	270
30	115	96	271	269
40	107	93	267	267
50	171	91	273	268
60	51	42	36	40
70	49	44	32	37
80	54	39	34	44
90	54	40	37	43

`nation` join `region` at 8KB page size

B	NLJ	BNLJ	HJ	SMJ
10	128	112	375	376
20	137	110	366	372
30	141	112	374	366
40	135	109	368	370
50	134	108	370	377
60	60	44	35	46
70	81	43	36	44
80	61	41	34	40
90	313	42	38	40

`nation` join `region` at 16KB page size

B	NLJ	BNLJ	HJ	SMJ
10	1149	1530	850	874
20	1281	645	276	521
30	1215	587	117	439
40	1138	592	111	438
50	1164	590	101	345
60	1145	585	98	261
70	1254	573	105	266
80	539	567	106	144
90	548	595	89	109

`nation` join `supplier` at 4KB page size

B	NLJ	BNLJ	HJ	SMJ
10	992	1210	952	981
20	1000	1227	961	985
30	1010	557	87	453
40	1019	560	88	452
50	1034	550	87	452
60	1022	567	85	335
70	1087	564	88	339
80	553	540	83	110
90	552	548	84	110

`nation` join `supplier` at 8KB page size

B	NLJ	BNLJ	HJ	SMJ
10	1043	1189	1370	1388
20	1037	1186	1364	1383
30	1030	1196	1366	1377
40	1072	542	89	392
50	1110	547	91	390
60	1093	548	91	393
70	554	545	87	114
80	554	538	85	110
90	554	548	79	102

`nation` join `supplier` at 16KB page size

B	NLJ	BNLJ	HJ	SMJ
10	16825	8936	1331	6206
20	16647	8723	1350	6093
30	16676	8686	1298	6019
40	16664	8593	1270	5803
50	16572	8563	1177	4685
60	16720	8548	1149	4368
70	16514	8403	1137	3488
80	16547	8312	1123	2973
90	16578	8351	1092	2180

`nation` join `customer` at 4KB page size

B	NLJ	BNLJ	HJ	SMJ
10	13161	8132	1126	5925
20	13171	8236	1095	5605
30	13237	8139	1036	5570
40	13409	8205	1030	5488
50	13270	8114	1002	5374
60	13229	8053	989	4134
70	13173	7999	965	3932
80	13367	7982	962	3100
90	13382	7935	958	2184

`nation` join `customer` at 8KB page size

B	NLJ	BNLJ	HJ	SMJ
10	11996	8081	1095	5897
20	12064	8040	1059	5502
30	12111	8016	1037	5316
40	12073	7987	991	5245
50	12040	8040	958	5189
60	11919	7900	957	5008
70	12167	7943	944	4980
80	12435	7872	931	4152
90	12156	7806	928	2755

`nation` join `customer` at 16KB page size

B	NLJ	BNLJ	HJ	SMJ
10	2309367	1360272	44375	303965
20	2355085	1357109	44205	306161
30	2410022	1337117	43492	282354
40	2433138	1322488	42650	252466
50	2414083	1338468	42131	226027
60	2417982	1311260	41376	195770
70	2404695	1299934	40731	162717
80	2410054	1292312	39971	132302
90	2426069	1288209	39356	102825

`supplier` join `lineitem`
at 4KB page size

B	NLJ	BNLJ	HJ	SMJ
10	1933412	1353066	40035	308372
20	1994737	1339779	40277	336686
30	2049400	1336416	39692	350331
40	2060376	1327760	39595	279714
50	2038850	1299728	38718	274310
60	2045270	1287989	38258	229802
70	2057047	1289750	37791	179385
80	2049774	1271385	37366	148257
90	2031252	1260389	36813	107184

`supplier` join `lineitem`
at 8KB page size

B	NLJ	BNLJ	HJ	SMJ
10	1807428	1336086	38261	377155
20	1851745	1339316	38547	414984
30	1925322	1318849	38208	438785
40	1930126	1307642	37941	446374
50	1924047	1297033	37369	439572
60	1921517	1281222	37256	306588
70	1921300	1269792	36562	296770
80	1899418	1257984	36269	196613
90	1892802	1247668	35758	133536

`supplier` join `lineitem`
at 16KB page size

A.2.2 Configuration with SSD as Persistent Storage Device

B	NLJ	BNLJ	HJ	SMJ
10	128	126	230	235
20	136	129	233	241
30	134	130	235	242
40	138	48	50	158
50	233	48	48	92
60	185	48	49	92
70	73	44	38	39
80	64	49	35	37
90	67	43	35	39

`nation` join `region` at 4KB page size

B	NLJ	BNLJ	HJ	SMJ
10	101	91	262	258
20	116	89	270	260
30	108	92	266	261
40	106	89	264	258
50	180	89	264	255
60	55	40	31	39
70	58	42	31	40
80	54	42	29	40
90	58	40	30	43

`nation` join `region` at 8KB page size

B	NLJ	BNLJ	HJ	SMJ
10	138	109	364	359
20	137	110	363	363
30	140	111	360	359
40	139	113	363	368
50	136	109	363	361
60	67	45	39	42
70	81	45	37	45
80	116	41	37	41
90	352	42	35	39

`nation` join `region` at 16KB page size

B	NLJ	BNLJ	HJ	SMJ
10	1129	1512	837	864
20	1295	644	269	515
30	1235	591	110	435
40	1138	588	109	434
50	1158	589	99	342
60	1141	581	97	259
70	1289	570	98	260
80	538	570	105	145
90	550	586	91	105

`nation` join `supplier` at 4KB page size

B	NLJ	BNLJ	HJ	SMJ
10	983	1200	934	962
20	979	1203	943	969
30	1006	556	85	444
40	1000	547	85	443
50	1017	556	86	441
60	1006	547	85	328
70	1092	541	83	328
80	540	540	79	105
90	534	536	83	105

`nation` join `supplier` at 8KB page size

B	NLJ	BNLJ	HJ	SMJ
10	1017	1175	1334	1353
20	1010	1176	1329	1335
30	1027	1178	1330	1349
40	1066	539	89	380
50	1096	540	86	381
60	1071	553	89	376
70	546	540	81	105
80	534	533	82	106
90	540	539	80	108

`nation` join `supplier` at 16KB page size

B	NLJ	BNLJ	HJ	SMJ
10	16660	8855	1317	6182
20	16408	8703	1333	5998
30	16454	8580	1301	5948
40	16472	8530	1270	5708
50	16359	8493	1172	4622
60	16538	8450	1141	4298
70	16301	8366	1123	3447
80	16337	8297	1125	2965
90	16366	8291	1084	2194

`nation` join `customer` at 4KB page size

B	NLJ	BNLJ	HJ	SMJ
10	13131	8049	1142	5801
20	13109	8212	1109	5490
30	13066	8130	1032	5452
40	13259	8147	1030	5388
50	13167	8067	1004	5290
60	13220	8080	987	4088
70	13119	8025	970	3883
80	13287	8029	957	3085
90	13354	7954	949	2176

`nation` join `customer` at 8KB page size

B	NLJ	BNLJ	HJ	SMJ
10	11889	8027	1077	5861
20	11931	8027	1060	5430
30	11987	8021	1058	5265
40	11949	7956	979	5171
50	12000	7974	963	5085
60	11896	7947	955	5005
70	12080	7928	941	4954
80	12187	7877	928	4103
90	12146	7843	927	2739

`nation` join `customer` at 16KB page size

B	NLJ	BNLJ	HJ	SMJ
10	2291349	1351129	44461	303907
20	2343411	1351197	44253	305275
30	2398598	1333049	43538	281500
40	2416858	1319223	42831	251851
50	2407897	1336714	42342	225861
60	2406668	1310711	41493	195348
70	2396081	1296501	40852	162543
80	2389467	1289886	40158	132431
90	2415131	1288438	39564	102790

`supplier` join `lineitem`
at 4KB page size

B	NLJ	BNLJ	HJ	SMJ
10	1907386	1349963	40082	304789
20	1972170	1332948	40155	333343
30	2026284	1315422	39660	345439
40	2034684	1308268	39290	274629
50	2032892	1293677	38695	271975
60	2034368	1279610	38252	228408
70	2036886	1285963	37831	178477
80	2030278	1268401	37242	147677
90	2029801	1257227	36837	106934

`supplier` join `lineitem`
at 8KB page size

B	NLJ	BNLJ	HJ	SMJ
10	1794881	1344462	38308	375200
20	1848384	1330418	38309	415212
30	1915251	1319057	38245	437938
40	1921327	1304844	37931	446052
50	1931368	1292379	37399	439785
60	1903610	1283551	37081	305940
70	1916993	1265438	36634	295770
80	1900039	1256841	36163	195706
90	1867559	1246512	35776	133216

`supplier` join `lineitem`
at 16KB page size

A.2.3 Configuration with SSD as Extension to the Buffer Pool

B	NLJ	BNLJ	HJ	SMJ
10	183	137	135	143
20	187	141	135	140
30	190	138	134	139
40	146	137	197	128
50	152	132	191	124
60	155	133	191	119
70	152	132	123	122
80	140	127	120	118
90	141	128	123	122
100	147	126	118	118

`nation` join `region` at 4KB page size and 10% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	167	128	123	126
20	176	129	123	128
30	196	131	124	133
40	180	134	192	124
50	206	130	192	128
60	221	131	189	128
70	217	137	126	120
80	201	136	128	125
90	223	137	125	124
100	217	133	127	126

`nation` join `region` at 4KB page size and 20% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	179	134	127	130
20	259	131	127	132
30	324	131	123	130
40	216	128	185	118
50	295	129	190	120
60	323	129	189	122
70	278	134	131	124
80	247	138	126	124
90	203	138	130	124
100	287	140	130	128

`nation` join `region` at 4KB page size and 30% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	149	51	40	48
20	189	49	38	50
30	266	48	40	50
40	330	48	38	48
50	331	48	42	50
60	292	49	41	51
70	226	51	30	48
80	236	50	30	48
90	287	52	30	46
100	299	50	31	44

`nation` join `region` at 4KB page size and 40% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	139	47	35	45
20	171	47	34	53
30	193	48	33	52
40	197	47	34	52
50	222	46	36	51
60	226	49	32	53
70	225	47	34	51
80	224	48	31	54
90	228	47	32	52
100	228	47	32	48

`nation` join `region` at 4KB page size and 50% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	169	43	42	47
20	245	44	37	48
30	299	45	39	47
40	321	48	33	55
50	309	48	34	51
60	329	48	34	52
70	298	48	36	51
80	311	47	34	50
90	317	48	34	54
100	319	49	31	55

`nation` join `region` at 4KB page size and 60% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	86	42	30	30
20	46	46	23	29
30	42	48	29	27
40	43	48	30	26
50	42	47	28	26
60	43	47	26	28
70	43	46	29	28
80	43	46	28	27
90	43	46	26	26
100	43	46	28	26

`nation` join `region` at 4KB page size and 70% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	61	43	32	31
20	44	49	28	26
30	42	48	31	24
40	42	49	28	25
50	42	48	27	25
60	42	48	27	26
70	42	49	29	24
80	43	48	30	23
90	44	49	30	24
100	45	46	28	25

`nation` join `region` at 4KB page size and 80% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	70	47	28	31
20	75	46	27	28
30	75	48	29	30
40	76	47	29	30
50	71	46	27	30
60	67	47	27	29
70	68	48	31	27
80	74	47	27	30
90	67	47	29	29
100	71	48	30	29

`nation` join `region` at 4KB page size and 90% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	108	92	154	140
20	121	93	156	138
30	121	92	154	136
40	122	91	153	135
50	120	92	145	131
60	126	90	154	134
70	121	90	154	134
80	123	91	153	133
90	116	91	101	106
100	117	94	98	106

`nation` join `region` at 8KB page size and 10% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	107	93	154	134
20	110	92	152	139
30	107	95	154	140
40	107	92	151	139
50	108	95	154	139
60	122	92	157	137
70	121	93	155	136
80	114	90	152	136
90	116	90	97	104
100	115	94	99	105

`nation` join `region` at 8KB page size and 20% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	106	93	147	132
20	104	92	146	134
30	105	95	153	136
40	104	96	152	136
50	114	92	154	136
60	120	90	152	137
70	114	94	95	104
80	112	92	96	105
90	114	92	95	105
100	113	94	97	105

`nation` join `region` at 8KB page size and 30% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	117	92	101	105
20	117	92	99	103
30	121	93	100	104
40	125	92	100	103
50	184	92	100	105
60	191	92	101	105
70	188	92	101	105
80	190	93	101	102
90	190	92	101	106
100	195	92	101	104

`nation` join `region` at 8KB page size and 40% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	119	92	101	104
20	117	89	102	104
30	129	90	101	101
40	134	92	99	102
50	194	91	99	98
60	212	90	90	100
70	228	92	93	101
80	237	91	99	105
90	226	94	101	104
100	207	91	102	105

`nation` join `region` at 8KB page size and 50% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	46	43	24	24
20	46	43	25	26
30	51	39	22	25
40	49	43	21	26
50	45	43	22	26
60	47	42	24	25
70	46	44	20	26
80	47	42	23	25
90	46	43	25	26
100	45	44	21	26

`nation` join `region` at 8KB page size and 60% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	48	38	24	26
20	43	46	25	25
30	45	46	28	27
40	46	42	24	26
50	45	43	23	27
60	44	44	25	24
70	44	45	26	26
80	48	42	22	25
90	46	40	21	23
100	47	41	20	25

`nation` join `region` at 8KB page size and 70% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	46	44	23	26
20	48	41	20	23
30	47	42	23	27
40	70	39	22	23
50	47	42	20	25
60	47	42	24	28
70	53	40	22	23
80	60	42	24	24
90	47	41	21	25
100	47	42	20	23

`nation` join `region` at 8KB page size and 80% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	61	39	22	27
20	48	42	21	28
30	46	42	23	24
40	46	42	23	24
50	47	45	23	24
60	58	40	21	24
70	50	41	23	24
80	48	40	20	24
90	48	42	24	24
100	53	39	22	23

`nation` join `region` at 8KB page size and 90% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	145	111	156	142
20	148	112	158	140
30	151	112	156	141
40	153	112	157	143
50	154	114	157	138
60	157	113	158	143
70	156	112	159	142
80	151	112	156	142
90	147	111	159	125
100	146	110	157	122

`nation` join `region` at 16KB page size and 10% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	132	112	158	144
20	135	113	156	142
30	146	112	156	139
40	155	111	156	142
50	156	111	157	142
60	161	112	158	138
70	154	111	155	137
80	143	109	149	137
90	146	110	159	126
100	143	108	158	126

`nation` join `region` at 16KB page size and 20% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	149	108	149	134
20	150	109	153	137
30	150	108	152	138
40	152	112	157	142
50	148	109	152	138
60	148	110	157	142
70	149	112	167	133
80	148	112	164	131
90	148	112	166	132
100	151	112	165	132

`nation` join `region` at 16KB page size and 30% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	148	112	164	128
20	148	110	159	127
30	151	111	161	127
40	154	109	160	128
50	149	107	155	120
60	159	113	163	131
70	157	111	164	129
80	159	112	162	127
90	157	112	163	127
100	155	112	164	126

`nation` join `region` at 16KB page size and 40% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	156	111	162	126
20	168	114	162	124
30	151	112	165	128
40	173	109	161	130
50	169	110	162	127
60	172	110	161	123
70	179	109	159	125
80	189	113	163	131
90	162	111	163	128
100	183	111	162	127

`nation` join `region` at 16KB page size and 50% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	56	41	29	23
20	57	41	27	25
30	60	41	32	25
40	56	42	27	27
50	56	42	28	26
60	55	42	27	26
70	58	42	31	29
80	59	42	32	28
90	58	42	30	30
100	54	42	31	30

`nation` join `region` at 16KB page size and 60% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	60	42	29	27
20	59	43	30	25
30	59	41	27	24
40	56	41	27	26
50	56	41	28	25
60	58	41	29	25
70	58	41	28	25
80	54	41	29	24
90	59	41	28	23
100	58	42	29	26

`nation` join `region` at 16KB page size and 70% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	65	41	30	25
20	95	40	31	30
30	99	41	29	29
40	61	41	32	30
50	57	41	29	26
60	81	39	29	24
70	127	40	28	30
80	76	40	31	28
90	103	40	31	30
100	110	40	31	30

`nation` join `region` at 16KB page size and 80% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	54	42	28	29
20	54	43	29	28
30	63	41	29	29
40	54	42	29	27
50	62	42	30	25
60	54	41	28	25
70	52	42	28	25
80	54	43	31	25
90	54	41	28	27
100	53	42	30	29

`nation` join `region` at 16KB page size and 90% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1698	1648	1050	3022
20	1703	1638	452	2264
30	1693	1631	371	1317
40	1702	1644	1684	16882
50	1666	1616	1035	2461
60	1208	1641	1696	16728
70	1199	1617	349	1144
80	1193	1605	645	2340
90	1181	1597	344	1108
100	1186	1598	348	1103

`nation` join `supplier` at 4KB page size and 10% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1685	693	407	625
20	1770	689	152	604
30	1751	679	286	583
40	1792	693	1162	2074
50	1316	688	593	715
60	1294	671	677	1404
70	1291	665	455	445
80	1269	665	485	423
90	1227	659	280	422
100	1236	663	281	421

`nation` join `supplier` at 4KB page size and 20% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1667	614	172	544
20	1736	609	149	460
30	1244	606	127	387
40	1324	616	546	1391
50	1275	607	312	968
60	1247	609	483	1273
70	1229	604	133	364
80	1259	606	162	397
90	1200	599	123	349
100	1215	602	113	336

`nation` join `supplier` at 4KB page size and 30% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1669	618	175	546
20	1852	611	146	459
30	1335	615	127	388
40	1253	617	128	389
50	1242	604	162	398
60	1259	599	140	367
70	1303	605	142	357
80	1341	611	150	355
90	1276	605	150	355
100	1258	605	152	355

`nation` join `supplier` at 4KB page size and 40% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1728	610	149	468
20	1472	599	147	346
30	1470	596	145	345
40	1418	596	143	344
50	1368	603	142	343
60	1306	599	144	345
70	1433	599	143	344
80	1437	598	142	344
90	1292	596	143	345
100	1298	596	143	345

`nation` join `supplier` at 4KB page size and 50% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1282	588	143	340
20	1305	581	139	339
30	1292	581	139	338
40	1305	584	139	337
50	1311	585	139	338
60	1298	586	139	337
70	1294	582	142	338
80	1317	589	140	337
90	1305	582	139	338
100	1315	582	139	339

`nation` join `supplier` at 4KB page size and 60% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1279	591	145	339
20	1486	586	140	337
30	1590	584	145	340
40	1572	584	144	339
50	1560	583	144	341
60	1590	588	144	340
70	1550	592	145	340
80	1430	587	145	340
90	1540	585	144	340
100	1595	586	146	342

`nation` join `supplier` at 4KB page size and 70% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	633	588	149	192
20	811	586	153	193
30	897	588	149	190
40	857	593	151	192
50	764	589	150	190
60	808	587	150	193
70	848	588	147	194
80	853	588	150	191
90	850	590	151	189
100	795	589	149	188

`nation` join `supplier` at 4KB page size and 80% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	542	577	144	181
20	547	579	143	179
30	548	575	147	184
40	551	577	142	178
50	555	577	141	180
60	548	580	142	184
70	545	577	145	184
80	547	579	141	178
90	554	580	141	183
100	548	583	140	179

`nation` join `supplier` at 4KB page size and 90% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1409	1265	517	2604
20	1409	1255	579	3976
30	1418	1253	506	2815
40	1418	1255	501	2798
50	1403	1253	505	2811
60	1025	1253	539	2699
70	1031	1252	538	2697
80	1024	1250	503	3104
90	1023	1252	504	2744
100	1029	1249	504	2732

`nation` join `supplier` at 8KB page size and 10% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1393	1260	550	3782
20	1404	1263	553	3780
30	1420	1266	505	2809
40	1415	1261	502	2799
50	1414	1253	499	2825
60	1022	1258	541	2734
70	1018	1252	541	2709
80	1024	1256	456	3017
90	1018	1247	499	2720
100	1023	1248	503	2738

`nation` join `supplier` at 8KB page size and 20% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1402	575	168	609
20	1418	573	168	607
30	1041	569	163	478
40	1045	565	164	473
50	1042	566	162	476
60	1038	562	163	475
70	1023	567	139	369
80	1025	565	141	369
90	1033	566	138	370
100	1027	561	138	369

`nation` join `supplier` at 8KB page size and 30% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1397	564	144	472
20	1401	565	143	471
30	1060	561	138	369
40	1058	566	137	369
50	1030	560	140	370
60	1032	564	139	371
70	1023	566	138	369
80	1029	566	138	369
90	1023	567	138	372
100	1019	565	138	369

`nation` join `supplier` at 8KB page size and 40% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1401	566	141	467
20	1401	568	142	470
30	1040	558	137	369
40	1053	565	138	369
50	1029	564	139	372
60	1031	561	138	370
70	1022	567	139	372
80	1032	568	139	369
90	1013	565	140	370
100	1043	562	141	370

`nation` join `supplier` at 8KB page size and 50% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1079	561	136	307
20	1067	569	138	309
30	1033	572	137	308
40	1035	560	140	307
50	1044	560	137	309
60	1043	560	137	308
70	1035	569	136	308
80	1035	560	140	310
90	1040	562	139	309
100	1036	562	139	310

`nation` join `supplier` at 8KB page size and 60% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1029	559	135	309
20	1032	568	137	306
30	1048	565	136	308
40	1035	563	137	309
50	1029	562	140	309
60	1034	568	136	308
70	1039	565	136	309
80	1058	563	138	308
90	1036	556	139	309
100	1043	566	136	307

`nation` join `supplier` at 8KB page size and 70% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	561	555	131	176
20	543	551	137	182
30	546	550	140	180
40	554	554	133	179
50	551	552	130	177
60	547	553	138	184
70	548	552	136	179
80	553	560	131	177
90	543	553	132	180
100	547	561	138	186

`nation` join `supplier` at 8KB page size and 80% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	549	554	132	177
20	546	552	134	180
30	544	553	134	179
40	545	547	139	182
50	547	550	134	180
60	551	554	136	179
70	542	557	132	178
80	548	553	135	178
90	548	548	137	181
100	544	554	134	181

`nation` join `supplier` at 8KB page size and 90% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1505	1219	606	2847
20	1515	1210	597	2823
30	1503	1217	601	2841
40	1056	1227	588	3131
50	1057	1209	586	3114
60	1043	1220	587	3126
70	1051	1215	592	3537
80	1049	1211	588	3517
90	1085	1231	593	3580
100	1086	1239	596	3620

`nation` join `supplier` at 16KB page size and 10% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1506	1207	605	2824
20	1499	1215	602	2816
30	1500	1210	598	2814
40	1036	1206	584	3101
50	1057	1205	584	3097
60	1048	1203	586	3089
70	1056	1212	586	3516
80	1057	1209	585	3493
90	1054	1209	582	3530
100	1050	1225	584	3547

`nation` join `supplier` at 16KB page size and 20% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1499	1211	596	2817
20	1500	1204	597	2819
30	1510	1207	597	2825
40	1050	1208	586	3111
50	1040	1214	584	3112
60	1054	1209	587	3113
70	1052	1218	586	3553
80	1047	1215	584	3566
90	1059	1222	590	3605
100	1052	1214	588	3586

`nation` join `supplier` at 16KB page size and 30% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1060	548	135	373
20	1071	543	136	373
30	1062	548	136	374
40	1074	552	137	373
50	1066	550	136	373
60	1056	546	136	372
70	1076	553	135	375
80	1066	551	136	375
90	1076	545	138	375
100	1074	544	136	374

`nation` join `supplier` at 16KB page size and 40% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1095	547	136	375
20	1095	546	136	372
30	1073	546	137	375
40	1082	557	136	375
50	1087	553	136	371
60	1078	549	136	372
70	1082	558	135	374
80	1085	553	135	375
90	1076	555	135	374
100	1091	557	135	376

`nation` join `supplier` at 16KB page size and 50% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1076	562	136	373
20	1077	554	134	372
30	1099	552	135	372
40	1089	563	137	374
50	1082	564	136	371
60	1090	560	138	373
70	1092	566	137	375
80	1084	559	137	374
90	1113	551	136	375
100	1082	556	138	375

`nation` join `supplier` at 16KB page size and 60% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	562	555	136	176
20	558	546	136	169
30	551	551	135	170
40	548	552	131	170
50	541	540	131	167
60	550	542	127	167
70	555	550	129	169
80	553	554	132	170
90	542	552	129	170
100	544	542	138	174

`nation` join `supplier` at 16KB page size and 70% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	542	537	137	175
20	555	544	125	166
30	550	554	130	170
40	545	553	127	166
50	549	543	127	166
60	539	541	129	168
70	552	542	129	171
80	549	548	130	172
90	548	546	130	172
100	545	543	133	170

`nation` join `supplier` at 16KB page size and 80% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	541	539	133	174
20	541	562	129	168
30	555	545	129	169
40	544	545	136	176
50	546	549	134	174
60	550	549	131	170
70	545	551	132	170
80	547	552	127	169
90	556	555	135	172
100	541	547	130	169

`nation` join `supplier` at 16KB page size and 90% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	24212	9036	1447	6899
20	23994	9156	1411	6410
30	24013	9102	1376	6413
40	23774	8852	1325	6455
50	24163	9010	1328	6382
60	23598	8805	1278	6632
70	24082	8896	1237	6380
80	24159	8784	1260	6293
90	17220	8995	1231	4163
100	17272	8965	1233	4157

`nation` join `customer` at 4KB page size and 10% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	24098	9337	1367	6461
20	23994	9298	1338	6448
30	23892	9282	1315	6448
40	24314	9089	1334	6320
50	24209	9230	1260	6403
60	24191	9124	1231	6376
70	23779	9224	1202	6290
80	17491	9158	1226	4207
90	17462	9186	1235	4202
100	17449	9189	1244	4200

`nation` join `customer` at 4KB page size and 20% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	24652	9429	1330	6490
20	24552	9406	1301	6441
30	24410	9447	1277	6450
40	24418	9309	1258	6364
50	24585	9308	1228	6380
60	24469	9326	1198	6293
70	17808	9357	1212	4206
80	17837	9411	1210	4193
90	17828	9379	1179	3808
100	17829	9363	1217	4202

`nation` join `customer` at 4KB page size and 30% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	24962	9652	1306	6409
20	24815	9652	1282	6339
30	24781	9629	1259	6360
40	24788	9586	1226	6326
50	24932	9545	1202	6286
60	18159	9514	1246	4239
70	18156	9576	1420	4142
80	18144	9606	1533	4282
90	18127	9568	1540	4291
100	18103	9571	1543	4291

`nation` join `customer` at 4KB page size and 40% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	25142	9670	1567	6271
20	24807	9658	1543	6241
30	24903	9656	1520	6188
40	24988	9568	1493	6157
50	18328	9642	1525	4260
60	18390	9661	1524	4285
70	18317	9665	1522	4281
80	18329	9660	1516	4282
90	18349	9660	1521	4298
100	18318	9651	1519	4288

`nation` join `customer` at 4KB page size and 50% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	25277	9696	1523	6238
20	25321	9652	1510	6188
30	25348	9650	1474	6150
40	18661	9671	1495	4271
50	18638	9681	1503	4275
60	18699	9691	1494	4273
70	18657	9698	1502	4276
80	18665	9693	1501	4272
90	18596	9703	1503	4262
100	18655	9692	1495	4251

`nation` join `customer` at 4KB page size and 60% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	25437	9684	1475	6200
20	25514	9644	1450	6150
30	18893	9721	1470	4226
40	18948	9737	1468	4235
50	18932	9734	1467	4219
60	18909	9713	1474	4250
70	18910	9735	1475	4242
80	18856	9710	1476	4249
90	18894	9710	1472	4255
100	18882	9719	1473	4240

`nation` join `customer` at 4KB page size and 70% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	25797	9653	1438	6138
20	19115	9756	1452	4234
30	19121	9762	1455	4234
40	19161	9723	1455	4254
50	19051	9704	1455	4236
60	19197	9764	1453	4240
70	19161	9748	1453	4245
80	19186	9740	1448	4235
90	19154	9742	1452	4246
100	19182	9722	1452	4254

`nation` join `customer` at 4KB page size and 80% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	19085	9619	1427	4208
20	19168	9661	1426	4244
30	19231	9633	1428	4221
40	19155	9660	1430	4222
50	19133	9652	1427	4246
60	19192	9629	1430	4253
70	19235	9625	1429	4224
80	19172	9653	1427	4251
90	19189	9622	1428	4244
100	19224	9641	1428	4233

`nation` join `customer` at 4KB page size and 90% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	17207	8462	1496	5310
20	17340	8768	955	4854
30	17385	8786	948	4833
40	17266	8677	930	4807
50	17313	8659	919	4827
60	17218	8678	898	4820
70	17295	8601	877	4752
80	17236	8576	868	4746
90	13329	8397	853	2703
100	13338	8411	851	2719

`nation` join `customer` at 8KB page size and 10% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	17277	8776	951	4902
20	17263	8804	929	4860
30	17385	8700	922	4826
40	17365	8715	912	4831
50	17398	8661	894	4832
60	17289	8664	879	4804
70	17145	8577	857	4781
80	13440	8453	921	3141
90	13448	8471	867	2736
100	13355	8467	893	2716

`nation` join `customer` at 8KB page size and 20% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	17500	8647	957	4893
20	17411	8726	939	4893
30	17530	8571	935	4877
40	17432	8569	919	4888
50	17525	8523	887	4866
60	17504	8513	885	4891
70	13570	8332	840	2837
80	13511	8404	1279	3507
90	13557	8415	1311	3588
100	13515	8384	1268	3521

`nation` join `customer` at 8KB page size and 30% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	17686	8771	1615	5067
20	17755	8766	1612	5091
30	17794	8661	1684	5081
40	17795	8663	1672	5062
50	17799	8596	1646	5071
60	13879	8541	1516	3931
70	13906	8501	1519	3939
80	13975	8527	1521	3944
90	13950	8523	1525	3947
100	13953	8511	1514	3936

`nation` join `customer` at 8KB page size and 40% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	17884	8814	1625	5077
20	17838	8769	1610	5062
30	17831	8682	1665	5051
40	17878	8643	1641	5051
50	13988	8525	1540	3943
60	13986	8525	1558	3948
70	14068	8499	1544	3966
80	14044	8533	1550	3957
90	14083	8534	1564	3967
100	14022	8505	1552	3965

`nation` join `customer` at 8KB page size and 50% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	17961	8732	1612	5064
20	17922	8708	1615	5088
30	17914	8708	1575	5066
40	14124	8500	1539	3943
50	14096	8478	1539	3936
60	14095	8481	1546	3942
70	14136	8482	1542	3941
80	14200	8502	1552	3971
90	14151	8475	1544	3947
100	14125	8475	1545	3949

`nation` join `customer` at 8KB page size and 60% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	18008	8712	1575	5003
20	18059	8720	1553	4974
30	14204	8470	1517	3909
40	14222	8441	1532	3908
50	14238	8423	1534	3919
60	14233	8432	1536	3912
70	14216	8437	1536	3926
80	14307	8435	1530	3921
90	14307	8503	1546	3913
100	14295	8478	1527	3917

`nation` join `customer` at 8KB page size and 70% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	18348	8643	1537	4995
20	14418	8428	1496	3922
30	14480	8438	1505	3919
40	14508	8457	1493	3918
50	14484	8458	1498	3912
60	14496	8450	1504	3945
70	14511	8460	1495	3936
80	14527	8471	1485	3932
90	14475	8457	1493	3934
100	14547	8467	1492	3942

`nation` join `customer` at 8KB page size and 80% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	14514	8385	1482	3923
20	14527	8381	1478	3929
30	14571	8403	1477	3936
40	14596	8401	1473	3935
50	14588	8431	1473	3937
60	14586	8416	1470	3928
70	14587	8409	1479	3947
80	14661	8410	1468	3946
90	14652	8421	1471	3946
100	14617	8390	1466	3953

`nation` join `customer` at 8KB page size and 90% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	15509	8373	900	2886
20	15487	8415	908	3200
30	15474	8490	898	3203
40	15559	8452	894	3183
50	15474	8379	869	3201
60	15505	8377	851	3186
70	15478	8315	832	3111
80	15438	8372	819	3080
90	11828	8121	834	2413
100	11777	8062	830	2399

`nation` join `customer` at 16KB page size and 10% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	15419	8408	911	3199
20	15454	8388	910	3197
30	15416	8339	883	3181
40	15418	8319	888	3152
50	15428	8310	886	3181
60	15473	8242	857	3148
70	15460	8217	813	3112
80	12172	8186	846	2471
90	12201	8193	853	2476
100	12206	8200	864	2470

`nation` join `customer` at 16KB page size and 20% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	15493	8330	886	3173
20	15435	8361	877	3170
30	15398	8347	857	3178
40	15569	8318	851	3166
50	15506	8314	836	3154
60	15439	8237	817	3151
70	12300	8270	1617	3827
80	12315	8224	1657	3878
90	12342	8274	1663	3890
100	12322	8233	1646	3899

`nation` join `customer` at 16KB page size and 30% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	15543	8268	1668	5399
20	15650	8179	1654	5420
30	15631	8227	1641	5398
40	15605	8167	1627	5391
50	15606	8125	1628	5364
60	12268	8235	1539	3919
70	12337	8160	1526	3945
80	12322	8212	1537	3932
90	12295	8178	1530	3962
100	12306	8192	1530	3949

`nation` join `customer` at 16KB page size and 40% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	15772	8215	1647	5405
20	15626	8158	1629	5423
30	15741	8190	1620	5402
40	15678	8143	1585	5402
50	12405	8140	1521	4006
60	12422	8093	1521	3994
70	12425	8132	1523	4021
80	12423	8096	1525	4016
90	12424	8122	1516	3998
100	12427	8098	1522	4059

`nation` join `customer` at 16KB page size and 50% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	15780	8140	1630	5388
20	15776	8074	1601	5381
30	15757	8047	1608	5245
40	12432	8070	1523	3951
50	12417	8055	1530	3958
60	12466	8050	1528	3966
70	12442	8067	1529	3950
80	12506	8075	1526	3945
90	12466	8060	1526	3954
100	12489	8066	1535	3938

`nation` join `customer` at 16KB page size and 60% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	15822	8032	1603	5392
20	15850	8001	1573	5264
30	12564	8088	1554	3912
40	12493	8079	1552	3899
50	12552	8051	1547	3923
60	12513	8088	1547	3936
70	12528	8162	1554	3933
80	12569	8120	1561	3920
90	12558	8064	1552	3951
100	12519	8120	1560	3949

`nation` join `customer` at 16KB page size and 70% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	15929	8025	1573	5240
20	12809	8114	1526	3949
30	12969	8095	1550	3947
40	13015	8041	1534	3968
50	12749	8063	1530	3948
60	12882	8164	1550	3996
70	12903	8100	1533	3944
80	12935	8051	1539	3990
90	12898	8130	1550	3981
100	12915	8052	1539	3978

`nation` join `customer` at 16KB page size and 80% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	13048	8003	1522	3929
20	13035	8010	1524	3942
30	12869	7984	1522	3971
40	13114	8008	1535	3980
50	13104	8010	1518	3999
60	13183	8009	1528	3953
70	13124	8013	1533	3992
80	13179	8031	1523	3983
90	13160	8024	1517	3975
100	13162	8005	1523	3981

`nation` join `customer` at 16KB page size and 90% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	4150202	2333628	54875	55927
20	3896810	2330313	53579	55553
30	4151848	2332263	52748	54603
40	4151395	2333140	51950	53763
50	3908669	2338884	51112	52923
60	3904772	2341604	50366	51864
70	4155782	2340642	49700	51229
80	3884867	2339224	48903	50365
90	3355098	2348498	48701	48987
100	3369150	2348989	48704	49013

`supplier` join `lineitem` at 4KB page size and 10% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	855557	3178659	61377	63850
20	887676	3182167	60675	63283
30	854753	3175287	60035	62356
40	874758	3184879	59289	61492
50	895020	3182472	58397	60588
60	864267	3180640	57595	59683
70	829254	3177935	56803	58672
80	37893	3185580	56114	57360
90	47861	3178986	56488	57323
100	81132	3179938	56487	57375

`supplier` join `lineitem` at 4KB page size and 20% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1813958	3903935	67172	69424
20	1814221	3916263	66353	68627
30	1816219	3924675	65579	68141
40	1846612	3898842	64968	67128
50	1832399	3912885	64059	66210
60	1803097	3898452	63290	65503
70	997921	3905071	62507	66988
80	1017957	3899577	63713	70607
90	1048793	3902389	63083	69530
100	1051634	3898554	63783	69723

`supplier` join `lineitem` at 4KB page size and 30% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	2789260	236033	71704	74526
20	2785374	238617	71073	73474
30	2775480	234320	70479	72718
40	2772731	223055	69661	71930
50	2781615	230156	68659	71080
60	1963655	228752	67955	73044
70	1982875	225006	69302	79879
80	2014398	229567	69945	81463
90	2014838	235378	69609	79560
100	2017221	227560	69191	77926

`supplier` join `lineitem`
at 4KB page size and
40% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	3755879	702714	76041	78729
20	3754215	703136	75150	77813
30	3759469	699884	74444	76946
40	3733980	702383	73570	76058
50	2931761	700085	72864	74663
60	2951510	691616	72904	74555
70	2975691	698459	72885	74619
80	2963704	693453	72784	74752
90	2962770	695123	72829	74622
100	2976816	696992	72795	74747

`supplier` join `lineitem`
at 4KB page size and
50% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	453128	1198157	80904	81389
20	441147	1201540	79984	80569
30	407650	1206577	79233	79589
40	3905719	1196919	78364	78084
50	3939654	1195792	78411	78152
60	3956587	1215861	78451	78277
70	3969657	1192557	78435	78084
80	3964596	1203202	78471	78311
90	3960124	1196070	78368	78179
100	3948770	1198896	78267	78151

`supplier` join `lineitem`
at 4KB page size and
60% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1388652	1564302	82984	83337
20	1365066	1563134	82136	82119
30	573003	1551526	80914	81140
40	594467	1548444	80930	80971
50	632079	1561261	81090	80776
60	636530	1551612	81045	80883
70	623061	1551691	81136	80986
80	627980	1546923	81210	80787
90	620573	1564131	81152	80972
100	623573	1562001	81206	80965

`supplier` join `lineitem`
at 4KB page size and
70% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	2344439	1790866	83610	83709
20	1575115	1740081	82629	82629
30	1575008	1814058	82315	82900
40	1616830	1795307	82601	82097
50	1621497	1778761	82711	82499
60	1625454	1769524	82679	82976
70	1619345	1795975	82280	82876
80	1603095	1821881	82804	82345
90	1602187	1782046	82627	82569
100	1611410	1756434	82799	83024

`supplier` join `lineitem`
at 4KB page size and
80% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	2569079	1944417	83229	83795
20	2587677	1896963	83103	82991
30	2623549	1927101	82796	83390
40	2639243	1925744	83106	83320
50	2631515	1926810	82694	83178
60	2625874	1925125	82891	83392
70	2617637	1929440	82767	83556
80	2609416	1908225	83060	83215
90	2613411	1954251	82793	83496
100	2643818	1897894	83048	83117

`supplier` join `lineitem`
at 4KB page size and
90% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	2918868	1837252	36448	43355
20	2914115	1828555	36549	56086
30	2938848	1833548	35898	73728
40	2934228	1834444	35318	76172
50	2925228	1831039	34924	75929
60	2925025	1831653	34407	75175
70	2938121	1837281	33892	74447
80	2893011	1836102	33333	73064
90	2456142	1839163	33103	57981
100	2465213	1839883	33043	58236

`supplier` join `lineitem`
at 8KB page size and
10% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	3434366	2235417	40163	82175
20	3444489	2236773	39782	81821
30	3439137	2237653	39265	81522
40	3437632	2233649	38790	81158
50	3464712	2232325	38413	80719
60	3445318	2231439	37897	79995
70	3400225	2233063	37281	78104
80	2951671	2234942	42283	50221
90	2968721	2235932	43931	80060
100	2995703	2232995	41775	75366

`supplier` join `lineitem`
at 8KB page size and
20% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	3940812	2595090	43213	85837
20	3954110	2599807	42821	85680
30	3951308	2599038	42283	85182
40	3959608	2598480	41805	84688
50	3950897	2596203	41328	84073
60	3907849	2600151	40871	82538
70	3467640	2600054	40413	66711
80	3483110	2599118	40389	67073
90	3514526	2599235	40444	67274
100	3523628	2601072	40366	67350

`supplier` join `lineitem`
at 8KB page size and
30% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	150604	2896575	45348	88592
20	133800	2896316	44999	88682
30	130275	2913347	44545	88012
40	124541	2905373	43964	87447
50	96880	2896037	43444	85846
60	3951347	2903922	43023	70531
70	3967274	2901786	43060	70884
80	3995452	2893825	43003	71109
90	4008214	2904165	43119	71372
100	4011977	2905897	43060	71454

`supplier` join `lineitem`
at 8KB page size and
40% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	613652	3145126	46980	90643
20	598576	3141348	46537	90528
30	608991	3143527	46047	89837
40	569250	3142369	45601	88484
50	127659	3140846	45098	73435
60	142764	3153614	45207	73829
70	176856	3146360	45106	73980
80	184545	3138783	45063	74328
90	179142	3146065	45164	74380
100	187912	3146672	45169	74600

`supplier` join `lineitem`
at 8KB page size and
50% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1071953	3349616	48750	91875
20	1071033	3350840	48318	91225
30	1051246	3361485	47728	90210
40	585586	3351323	47273	75312
50	600262	3352000	47312	75691
60	634991	3358971	47243	76041
70	650099	3351098	47332	76236
80	651744	3349684	47340	76514
90	653412	3357813	47376	76442
100	653897	3350873	47248	76637

`supplier` join `lineitem`
at 8KB page size and
60% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1530049	3520187	49481	92259
20	1522211	3486369	49018	91460
30	1058780	3514220	48303	76638
40	1078413	3505433	48509	76703
50	1104479	3483072	48230	77359
60	1120343	3522804	48416	77264
70	1129462	3483596	48583	77470
80	1123945	3486884	48391	77959
90	1123368	3517677	48640	77797
100	1118206	3482777	48524	78373

`supplier` join `lineitem`
at 8KB page size and
70% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1988635	3623747	49727	91566
20	1498801	3618395	48956	77452
30	1522877	3578160	49229	77398
40	1571056	3628828	49072	77740
50	1589576	3593902	49066	78351
60	1595637	3586191	49274	77982
70	1599515	3632046	49026	78673
80	1590066	3590020	49323	79121
90	1594143	3606774	49271	78635
100	1597457	3622278	49123	79239

`supplier` join `lineitem`
at 8KB page size and
80% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	1945417	3661780	49261	77097
20	1969920	3653175	49192	77648
30	2011557	3684383	49332	78043
40	2036645	3680528	49162	78652
50	2045337	3660085	49296	79051
60	2063138	3654572	49421	79019
70	2057997	3684018	49339	78930
80	2051560	3680689	49104	79410
90	2061325	3664617	49367	79530
100	2045794	3653970	49319	79407

`supplier` join `lineitem`
at 8KB page size and
90% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	2454826	1600190	34430	57865
20	2465909	1592232	34085	57545
30	2469416	1593175	33776	57442
40	2479128	1600627	33339	57002
50	2469815	1593763	32936	56466
60	2476392	1593896	32456	56184
70	2475298	1592926	31962	55431
80	2436662	1594566	31530	54316
90	2078514	1594607	31310	45704
100	2078840	1594200	31278	45891

`supplier` join `lineitem`
at 16KB page size and
10% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	2736596	1777137	36108	60577
20	2735908	1775645	35720	60238
30	2738442	1773796	35298	60070
40	2744381	1775154	34906	59360
50	2742256	1773943	34508	59095
60	2741611	1779462	34045	58440
70	2698997	1780244	33596	57130
80	2327457	1785439	33392	48182
90	2343064	1783353	33411	48253
100	2373279	1782153	33405	48431

`supplier` join `lineitem`
at 16KB page size and
20% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	2985985	1956567	37232	62093
20	2999638	1959270	36833	61743
30	2999537	1963228	36544	61437
40	3006852	1963829	36105	61347
50	2995593	1959913	35679	60763
60	2984536	1960664	35239	59581
70	2596982	1958678	34963	50403
80	2613969	1958338	34948	50602
90	2648604	1956705	34946	50775
100	2658194	1955604	34929	50922

`supplier` join `lineitem`
at 16KB page size and
30% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	3247017	2093166	38218	63698
20	3246776	2098004	37742	63385
30	3243797	2096354	37396	62888
40	3235773	2097874	36964	62239
50	3210510	2097494	36529	61153
60	2845869	2099051	36274	52552
70	2858207	2099225	36323	52725
80	2891883	2096748	36290	52864
90	2902773	2100475	36273	52973
100	2908503	2098917	36242	53244

`supplier` join `lineitem`
at 16KB page size and
40% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	3513596	2231843	39056	64617
20	3518772	2230316	38675	64030
30	3503091	2226443	38228	63462
40	3462715	2231885	37828	62529
50	3114532	2231795	37589	54068
60	3129831	2228573	37629	54254
70	3160226	2226850	37569	54423
80	3167653	2231641	37535	54446
90	3169202	2232559	37530	54625
100	3171383	2228123	37497	54746

`supplier` join `lineitem`
at 16KB page size and
50% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	3734095	2316295	39406	64812
20	3727987	2324690	39070	64315
30	3716747	2327018	38670	63277
40	3349895	2319506	38231	54781
50	3364579	2332387	38305	55120
60	3401487	2321048	38128	55016
70	3406325	2321039	38288	55244
80	3410633	2331040	38340	55445
90	3409873	2318491	38290	55511
100	3402879	2325222	38439	55663

`supplier` join `lineitem`
at 16KB page size and
60% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	3945184	2400378	39682	64702
20	3946875	2398426	39231	63623
30	3562999	2400449	39083	55613
40	3579761	2399478	38949	56021
50	3617147	2397408	38980	56230
60	3635486	2399694	38870	56324
70	3638720	2397069	39054	56387
80	3633248	2396898	39094	56362
90	3634892	2398795	39171	56659
100	3637149	2400855	39029	56507

`supplier` join `lineitem`
at 16KB page size and
70% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	3900611	2444456	39374	63808
20	3762914	2442661	39196	55326
30	3792382	2443398	39157	55771
40	3832137	2429664	39100	56372
50	3858806	2440780	39285	56503
60	3867904	2445170	39309	56298
70	3864455	2442711	39251	56605
80	3868355	2438086	39230	56720
90	3864619	2444863	39393	56744
100	3866148	2443553	39240	56752

`supplier` join `lineitem`
at 16KB page size and
80% available RAM

B	NLJ	BNLJ	HJ	SMJ
10	3983483	2463655	38804	55134
20	3997131	2455954	38978	55690
30	4030771	2458543	38885	56008
40	4062910	2464163	39139	56055
50	4077626	2463065	39009	56427
60	4081522	2458799	39047	56720
70	4085109	2462238	38979	56505
80	4082053	2463622	39033	56570
90	4089304	2456749	38846	56766
100	4087736	2463191	39030	56928

`supplier` join `lineitem`
at 16KB page size and
90% available RAM

A.2.4 In-Memory Configuration

PS	NLJ	BNLJ	HJ	SMJ
4KB	40	38	23	32
8KB	39	37	23	29
16KB	38	38	25	33

`nation` join `region`

PS	NLJ	BNLJ	HJ	SMJ
4KB	571	565	74	104
8KB	566	537	77	110
16KB	568	534	77	110

`nation` join `supplier`

PS	NLJ	BNLJ	HJ	SMJ
4KB	8545	8503	830	1227
8KB	8660	8331	821	1215
16KB	8523	8057	820	1206

`nation` join `customer`

PS	NLJ	BNLJ	HJ	SMJ
4KB	1344499	1331276	33914	63643
8KB	1338135	1316403	33512	62925
16KB	1305419	1286363	33428	62008

`supplier` join `lineitem`

Bibliography

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer-Verlag, 2013. (cited on Page 56, 57, and 58)

[AMDM07] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. Materialization Strategies in a Column-Oriented DBMS. In *Proceedings of the International Conference on Data Engineering*, ICDE, pages 466–475. IEEE, 2007. (cited on Page 33 and 34)

[APW+08] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Technical Conference*, June 2008. (cited on Page 26 and 27)

[Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Conference on Software Product Lines*, SPLC, pages 7–20, Berlin, Heidelberg, 2005. Springer-Verlag. (cited on Page 32)

[BBR+12] Sebastian Breß, Felix Beier, Hannes Rauhe, Eike Schallehn, Kai-Uwe Sattler, and Gunter Saake. Automatic Selection of Processing Units for Coprocessing in Databases. In *Advances in Databases and Information Systems, ADBIS*, volume 7503 of *Lecture Notes in Computer Science*, pages 57–70. Springer-Verlag, Berlin, Heidelberg, 2012. (cited on Page 25)

[BBR+13] Sebastian Breß, Felix Beier, Hannes Rauhe, Kai-Uwe Sattler, Eike Schallehn, and Gunter Saake. Efficient Co-Processor Utilization in Database Query Processing. *Information Systems*, 38(8):1084–1096, 2013. (cited on Page 35)

[BFJ+86] Thomas Burns, Elizabeth Fong, David Jefferson, Richard Knox, Leo Mark, Christopher Reedy, Louis Reich, Nick Roussopoulos, and Walter Truszkowski. Reference Model for DBMS Standardization. *SIGMOD Record*, 15(1):19–58, March 1986. (cited on Page 5)

[BKM08] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the Memory Wall in MonetDB. *Communications of the ACM*, 51(12):77–85, December 2008. (cited on Page 22 and 23)

[BMK99] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Optimizing Database Architecture for the New Bottleneck: Memory Access. *Proceedings of the International Conferenece on Very Large Databases, VLDB*, 9(3):231–246, December 1999. (cited on Page 12, 13, 21, 34, 36, 37, and 40)

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–636, September 2010. (cited on Page 32 and 34)

[Den11] Yuhui Deng. What is the Future of Disk Drives, Death or Rebirth? *ACM Computing Surveys*, 43(3):23:1–23:27, April 2011. (cited on Page 12)

[DH96] Karel Driesen and Urs Hölzle. The Direct Cost of Virtual Function Calls in C++. *SIGPLAN Notices*, 31(10):306–323, October 1996. (cited on Page 50 and 53)

[DP09] Jaeyoung Do and Jignesh M. Patel. Join Processing for Flash SSDs: Remembering Past Lessons. In *Proceedings of the International Workshop on Data Management on New Hardware*, DaMoN, pages 1–8, New York, NY, USA, 2009. ACM. (cited on Page 26 and 27)

[FHL+07] Rui Fang, Bingsheng He, Mian Lu, Ke Yang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. GPUQP: Query Co-Processing Using Graphics Processors. In *Proceedings of the International Conference on Management of Data, SIGMOD*, pages 1061–1063, New York, NY, USA, 2007. ACM. (cited on Page 1, 24, and 37)

[FKA+13] Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. Do Background Colors Improve Program Comprehension in the #ifdef Hell? *Empirical Software Engineering*, 18(4):699–745, 2013. (cited on Page 58)

[GBS+12] Alexander Grebhahn, David Broneske, Martin Schäler, Reimar Schröter, Veit Köppen, and Gunter Saake. Challenges in Finding an Appropriate Multi-Dimensional Index Structure with Respect to Specific Use Cases. In *Grundlagen von Datenbanken*, pages 77–82, 2012. (cited on Page 46)

[GLW+04] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast Computation of Database Operations Using Graphics Processors. In *Proceedings of the International Conference on Management of Data, SIGMOD*, pages 215–226, New York, NY, USA, 2004. ACM. (cited on Page 1 and 24)

[Got75] Leo R. Gotlieb. Computing Joins of Relations. In *Proceedings of the International Conference on Management of Data, SIGMOD*, pages 55–63, New York, NY, USA, 1975. ACM. (cited on Page 13)

[Gra94] Goetz Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994. (cited on Page 23 and 34)

[Gra07] Goetz Graefe. The Five-Minute Rule Twenty Years Later, and How Flash Memory Changes the Rules. In *Proceedings of the International Workshop on Data Management on New Hardware, DaMoN*, pages 6:1–6:9, New York, NY, USA, 2007. ACM. (cited on Page 27 and 28)

[Här87] Theo Härder. Realisierung von operationalen Schnittstellen. In Peter C. Lockemann and Joachim W. Schmidt, editors, *Datenbank-Handbuch*, chapter 3, pages 163–335. Springer-Verlag, Berlin, Heidelberg, 1987. (cited on Page 5, 6, and 8)

[HCLS93] Laura M. Haas, Michael J. Carey, Miron Livny, and Amit Shukla. SEEKing the Truth about Ad Hoc Join Costs. *VLDB Journal*, 6:241–256, 1993. (cited on Page 15)

[HYF+08] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational Joins on Graphics Processors. In *Proceedings of the International Conference on Management of Data, SIGMOD*, pages 511–524, New York, NY, USA, 2008. ACM. (cited on Page 1, 24, 25, and 38)

[KLMV12] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. GPU Join Processing Revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware, DaMoN*, pages 55–62, New York, NY, USA, 2012. ACM. (cited on Page 1 and 24)

[KSC+09] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *Proceedings of the VLDB Endowment*, 2(2):1378–1389, 2009. (cited on Page 24 and 37)

[KSS12] Veit Köppen, Gunter Saake, and Kai-Uwe Sattler. *Data Warehouse Technologien*. MITP, 1. edition, August 2012. (cited on Page 22 and 28)

[Lar13] Paul Larson. Evolving the Architecture of a DBMS for Modern Hardware. In *Datenbanksysteme für Business, Technologie und Web, BTW*, volume 214 of *LNI*, page 19. GI, 2013. (cited on Page 1)

[Lei12] Thomas Leich. *Variables Nanodatenmanagement für eingebettete Systeme*. PhD thesis, University of Magdeburg, November 2012. (cited on Page 1)

[LKC+10] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. *SIGARCH Computing Architecture News*, 38(3):451–460, June 2010. (cited on Page 24 and 25)

[LSS13] Andreas Lübcke, Martin Schäler, and Gunter Saake. Dynamic Relational Data Management for Technical Applications. Technical Report 2, University of Magdeburg, 2013. (cited on Page 33)

[MBC09] Marcílio Mendonça, Moises Branco, and Donald D. Cowan. S.P.L.O.T.: Software Product Lines Online Tools. In *Proceedings of the Conference Companion on Object-Oriented Programming, Systems, Languages, and Applications, SIGPLAN*, pages 761–762, New York, NY, USA, 2009. ACM. (cited on Page 38)

[MBK02] Stefan Manegold, Peter Boncz, and Martin L. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. In *Proceedings of the International Conference on Very Large Databases, VLDB*, pages 191–202. VLDB Endowment, 2002. (cited on Page 46)

[ME92] Priti Mishra and Margaret H. Eich. Join Processing in Relational Databases. *ACM Computing Surveys*, 24(1):63–113, March 1992. (cited on Page 13, 14, 15, 16, 17, 18, 38, and 68)

[Mey05] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*, volume 3. Addison-Wesley Professional, 2005. (cited on Page 53)

[MT09] Rene Mueller and Jens Teubner. FPGA: What's in it for a Database? In *Proceedings of the International Conference on Management of Data, SIGMOD*, pages 999–1004, New York, NY, USA, 2009. ACM. (cited on Page 25, 26, and 35)

[PSG08] Milo Polte, Jiri Simsa, and Garth Gibson. Comparing Performance of Solid State Devices and Mechanical Disks. In *Proceedings of the Petascale Data Storage Workshop*, Austin, TX, 2008. (cited on Page 11)

[RDFH12] Philipp Rösch, Lars Dannecker, Franz Färber, and Gregor Hackenbroich. A Storage Advisor for Hybrid-Store Databases. *Proceedings of the VLDB Endowment*, 5(12):1748–1758, August 2012. (cited on Page 22)

[SGG+99] Steven W. Schlosser, John Linwood Griffin, John Linwood Grin, David F. Nagle, and Gregory R. Ganger. Filling the Memory Access Gap: A Case for On-Chip Magnetic Storage. Technical report, School of Computer Science, Carnegie Mellon University, 1999. (cited on Page 13)

[SGS+13] Martin Schäler, Alexander Grebhahn, Reimar Schröter, Sandro Schulze, Veit Köppen, and Gunter Saake. QuEval: Beyond High-Dimensional Indexing à la Carte. *PVLDB*, 6(14), 2013. accepted for publication 01/08/13. (cited on Page 46)

[Sha86] Leonard D. Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM Transactions on Database Systems, TODS*, 11(3):239–264, August 1986. (cited on Page 13)

[SKN94] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of the International Conference on Very Large Databases, VLDB*, pages 510–521, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc. (cited on Page 23)

[SMA+07] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *Proceedings of the International Conference on Very Large Databases, VLDB*, pages 1150–1160. VLDB Endowment, 2007. (cited on Page 1)

[SSH11] Gunter Saake, Kai-Uwe Sattler, and Andreas Heuer. *Datenbanken – Implementierungstechniken*. MITP, Bonn, 3 edition, Oktober 2011. (cited on Page 5, 6, 7, 9, 11, 12, 13, 16, 22, 34, 40, and 46)

[SSH13] Gunter Saake, Kai-Uwe Sattler, and Andreas Heuer. *Datenbanken. Konzepte und Sprachen*. MITP, Bonn, 5 edition, March 2013. (cited on Page 8 and 13)

[STSS13] Reimar Schröter, Thomas Thüm, Norbert Siegmund, and Gunter Saake. Automated Analysis of Dependent Feature Models. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS*, pages 9:1–9:5, New York, NY, USA, 2013. ACM. (cited on Page 37)

[TKB+13] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming*, 2013. (cited on Page 38)

[VL11] Pranav Vaidya and Jaehwan John Lee. A Novel Multicontext Coarse-Grained Reconfigurable Architecture (CGRA) For Accelerating Column-Oriented Databases. *ACM Transactions on Reconfigurable Technology and Systems*, 4(2):13:1–13:30, May 2011. (cited on Page 26)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 19.08.2013