

# Using Aspect-Orientation to Manage Database Statistics

Uwe Hohenstein

CT SE 2  
Siemens AG  
Otto-Hahn-Ring 6  
81730 München  
Uwe.Hohenstein@siemens.com

**Abstract:** Database-internal statistics are very important for cost-based query optimizers. Only if an optimizer has accurate statistical information about data, it provides excellent execution plans for queries; if those statistics are out of date, then queries will be badly optimized and performance suffers. Unfortunately, a lot of database systems currently do not manage statistics adequately; they only offer means to let statistics be calculated, but it is up to the user to decide when to initiate the calculation. In fact, this is a critical task because the calculation takes some minutes and should only be done if really necessary.

We present an effective approach that shifts the responsibility for managing the calculation of database statistics from the database system to applications. We show that aspect-orientation helps us to provide some automatic mechanism that concentrates the logic in one single aspect. This aspect decides for any application individually when to initiate a calculation of statistics for which tables.

## 1 Introduction

Database applications often suffer from a bad performance. Relational database systems (RDBSs) land themselves in it, as SQL is designed to allow for an easy access to databases, by providing powerful query capabilities. Hence, users can formulate complex SQL statements, joining several tables and filtering records. Internally, a RDBS's query optimizer creates an execution plan for each query. In fact, researchers, e.g., [OL90,Gr93,HC97], spent a lot of effort on query optimization [GLS93].

Besides some newer special-purpose proposals [ViN02,IN92], there are two approaches for building execution plans [Ch98] in RDBSs: Rule-based (RBO) and cost-based optimization (CBO). RBO has been introduced at the beginning of relational technology: It uses some fixed rules and essentially determines what existing indexes to use. CBO also takes into account the amount of data in tables and indexes as well as the selectivity of attributes [ChS95]. RDBS vendors recommend using CBO because it is the more recent technology and provides better execution plans. Moreover, newer concepts such as partitioned tables or star queries [GHQ95,TK02] can only be optimized with CBO.

It is surprising that the performance of database applications is sometimes bad in spite of powerful CBO technology. Thereby, it is often overlooked that CBO requires input:

precise database statistics about the amount of data in tables, the selectivity of attributes etc. [Ch98]. If these statistics are not up-to-date, performance could suffer dramatically. Despite being essential for CBO, recent RDBSs often do not compute data statistics automatically; they only offer means to initiate a calculation. Hence, the user is responsible for keeping them up-to-date by explicitly requesting a calculation. The most often used approach is to update the database statistics periodically, e.g., by a background job overnight when the load on the database is low. This does not work well if data is changing too frequently, quickly, and dramatically. A much higher frequency is then requested, which produces too heavy load. Calculating statistics can also be done before starting an application. Since updating database statistics takes several minutes up to one hour depending on the amount of data, the real application is delayed. Furthermore, calculating statistics before or after an application is not enough if massive data insertions and deletions are combined alternately with complex queries within an application. And there is no coordination between several users: Even if statistics are up-to-date, other users calculate again, because they are not aware of available statistics.

We were confronted with these problems in a real telecommunication scenario. Several applications read data from the database by using lots of complex queries, the performance of which suffers heavily from time to time – to be more precise, whenever the database statistics are stale: Then, complex queries run several minutes, but could run in milliseconds, if statistics are up-to-date. Unfortunately, calculating the complete statistics takes up to 60 minutes. Using estimated statistics, this time can be reduced, but only with an aggressive estimation percentage of less than 10%; this let several complex queries being still badly optimized. The performance problems are astonishing because the total amount of data is not huge, and the database schema is well-designed.

Vendors such as DB2 UDB [AH+04] are currently investigating solutions for automatically keeping statistics up-to-date, but solutions are not in all products so far. We propose a different approach in this paper that relieves a RDBS from this difficult task. Initiating statistics calculation of relevant tables becomes part of the application. The advantage is that statistics are managed for queries of an application individually. In fact, an application knows when complex queries are performed on what tables, when up-to-date statistics are required for these tables. That is, our approach can better be adjusted to applications and provides more flexibility than a centralized database-internal approach.

In Section 2, we present the basic idea of managing statistics, especially handling only relevant tables that should be subject to statistics calculation. Section 3 describes how the approach can benefit from the recent software technology of aspect-orientation (AO). AO aims at providing systematic means for the modularization of crosscutting concerns and shows benefit in various areas [KG02,RC03]. Crosscutting concerns are functionalities that are usually scattered around the code [EFB01]. Indeed, requesting up-to-date statistics is a crosscutting concern that affects several places in an application. We use AspectJ [Ki01], a general-purpose aspect-oriented extension of Java, to implement an effective application-specific management of database statistics. We benefit from one important AO feature: To extend existing code in a modular manner, to let applications here calculate statistics adequately without explicitly changing the application's code. In Section 4, we summarize the benefits of our approach for the telecommunication application.

## 2 Basic Idea

Finding the right place for enforcing the calculation of database statistics is not as easy as it seems to be. We are faced with the problem that up-to-date statistics are necessary for performing complex queries efficiently on the one hand. But on the other hand, calculating statistics can last several minutes, since it is not done incrementally: Every calculation uses the complete data of a table (and associated indexes) ignoring previously calculated statistics. It is up to recent research to find solutions for an incremental estimation [GMP97]. Certainly, statistics can be estimated to reduce the time, but often this does not work. In fact, [HN95,PI97] show that it is difficult to estimate the accurate number of distinct values.

An adequate approach must provide up-to-date database statistics for applications only when needed (in order not to delay an application), and on time for query execution. Since analyzing the statistics for a lot of tables takes some minutes, it is also necessary to reduce the amount of tables for collecting database statistics.

This can be achieved by some kind of environment. The environment mainly consists of a class CBO that should enable an application to mark those tables the statistics of which must be up-to-date, because a complex SQL query is based on those tables. This is the task of a *Request(List tables)* method. An application should call *Request* whenever it performs SELECT, DELETE or UPDATE queries with complex search conditions that require the database statistics to be up-to-date. The relevant tables are passed as *tables* parameter. In order to reduce the time by computing only the statistics of relevant tables, the method first checks whether the statistics for the passed tables are stale, i.e., that have changed since the last calculation. If there are no stale tables, true is immediately returned to the caller, because the statistics are up-to-date. Otherwise, statistics are calculated. This avoids useless calculations for already up-to-date tables, for example, if a *Request* for a query occurs in a loop.

Unfortunately, getting stale tables from a RDBS has generally some problems. For example in Oracle, all relevant tables must be defined with a MONITORING property at first. This is no problem since this property can be added to existing tables by means of an ALTER TABLE statement. In terms of performance, the overhead seems to be low because this information is kept in the SGA cache. There is a system view ALL\_TAB\_MODIFICATIONS that contains information about all the changes in monitored tables, but this information is only updated every three hours in Oracle8i and every 15 minutes in recent releases. The DBS\_STATS PL/SQL package also provides means to ask for stale statistics, but is based upon the ALL\_TAB\_MODIFICATIONS view. Hence, the information again is not up-to-date. Other RDBSs have similar problems.

A general possibility is to compare the real number of records in a table (SELECT COUNT(\*)) with the number of records the RDBS has noticed (in the system views). But such a check is too expensive for the purpose of checking staleness.

Consequently, we have to take other sources for staleness. An intuitive proposal is to let applications provide information about staleness. The proposed CBO class could offer a *Notify(List tables)* method for letting applications indicate larger modifications of data;

the affected tables are passed as tables parameter. The list of tables is only dumped into a private *requests* data member. Having the information given by *Notify*, the semantics of *Request* is now to calculate only the statistics of those tables that are passed to *Request* as parameter and that occur in the *requests* data member (filled by *Notify*).

This approach seems to be intuitive and reasonable. But the approach must be used appropriately in order not to run into bad performance behavior again. For example, if a *Notify* about larger table modifications is missing, the environment gets the impression that there are no stale tables, and consequently no calculation of statistics might take place. If a *Request* for a table is missing, then complex queries will run longer in case that statistics are not up-to-date. If a *Notify* is wrongly called for table, e.g., a table that has not changed much, the table is considered stale. This presumably causes unnecessary calculations in *Request* later on. Then statistics calculation takes place without any need, but leads to a heavy delay of applications. Moreover, the control flow must be carefully analyzed. We must know what parts of the code, particularly what SQL statements, are really executed, what loops are executed how often, what IF cases are relevant.

Even if we know that an SQL statement is performed, e.g., an UPDATE, we do not know how many records are affected. Should *Notify* be called or is it not necessary because only a few records are affected? Even if only one record is updated, this could happen in a loop that is executed a thousand times. Then it is worth to notify the environment about the table change after the loop. That is, we should take into account the amount of changed data. The SQL statement can certainly be asked, but this information must then be actively evaluated.

### 3 An Aspect-Oriented Approach

As we have seen, an approach for managing database statistics must be handled carefully. We now show an approach taking benefit from aspect-orientation (AO). We here assume that applications are written in Java using JDBC for database accesses.

The evolution of software development techniques has been driven by the need to achieve a better separation of crosscutting concerns (CCCs). CCCs are those functionalities that are typically spread across several classes. They lead to lower programming productivity, poor quality and traceability, lower degree of code reuse, and the lack of supporting evolution [La03]. AO employs special constructs to separate crosscutting concerns through the notion of an aspect. This separation allows for a better modularization, thereby avoiding the well-known symptoms of non-modularization such as code tangling and code scattering.

We use AspectJ [Ki01], which is one popular representative of an AO language extending Java. Most important for our purpose is that AspectJ allows one to add behavior to existing code *without* changing the code. AspectJ helps us to trap any execution of a relevant JDBC statement and to extract the SQL statements to be performed. This is done in a single module. Moreover, we can get the number of modified records. Hence, it is possible to detect any major changes in tables, and to insert a *Notify* after INSERT, DELETE and UPDATE statements only if a certain

threshold of modified records is passed. Similarly, AspectJ let us detect any SELECT, DELETE and UPDATE with a complex WHERE-part; a *Request* can be added to enforce up-to-date statistics before. The advantages are obvious:

- There is no manual activity; any misuse of *Notify/Request* is excluded. Consequently, we can rely upon collected statistics.
- The coding can be done outside the existing application code by means of adding aspects: Aspects work on all the code without touching that code.

Let us now dive into the technical details. Programming with AspectJ is essentially done by ordinary Java objects and newly *aspects*. Aspects are special units that crosscut objects and define some crosscutting functionality. The main purpose of aspects is to change the dynamic structure of a program by modifying the program flow. An aspect can intercept certain points of the program flow, called *join points*. A join point selects points of execution where to introduce crosscutting aspect code. Examples of join points are method calls or executions, constructor executions, and attribute accesses. The following aspect `UpdateStatistics` specifies join points for those JDBC operations that are of interest for us. It also introduces an internal hashmap `changeCounters` in order to maintain a record counter for each table.

```
public aspect UpdateStatistics {
    // manage counter for each table:
    HashMap changeCounters = new HashMap(); // <tabName,cnt>

    public pointcut jdbcExecuteQuery(String str) : // direct execute
        call(* java.sql.Statement+.executeQuery(String)) && args(str);
    public pointcut jdbcExecuteUpdate(String str) : // direct update
        call(* java.sql.Statement+.executeUpdate(String))&& args(str);
    public pointcut jdbcExecutePrepare(PreparedStatement stmt) :
        // execute prepared statement stmt
        (call(* java.sql.PreparedStatement.executeUpdate()) |
         call(* java.sql.PreparedStatement.executeQuery() )
         && target(stmt);
        ...
}
```

This aspect declares *pointcuts* to specify what join points should be trapped in the program flow. Pointcuts define the signature of relevant join points, i.e., JDBC statements we are interested in. Pointcut `jdbcExecuteQuery` specifies any immediate invocation of a query passed as a string, e.g., `stmt.executeQuery("SELECT ...")`: The expression `call(* java.sql. Statement+.executeQuery(String))` traps any such call of `executeQuery` on an object of class `Statement` in package `java.sql` with a `String` parameter, i.e., the query to be performed, and having any return type; wildcard `*` can be used to represent any class. Another wildcard `+` denotes that the objects can be of class `Statement` or of any subclass. Since we need information about the passed query string, we have to bind a variable `str` to the parameter value by means of `args(str)`. `str` can be used in advices later on.

Similarly, `jdbcExecuteUpdate` specifies any immediate invocation of a manipulation (UPDATE, DELETE, INSERT), i.e., to trap invocations of the form `stmt.executeUpdate("DELETE ...")`.

In order to speed up performance, so-called prepared statements are often used in JDBC:

```
PreparedStatement pstmt = conn.prepareStatement("UPDATE ...");
pstmt.setInt(1,9); pstmt.setString(2,"abc"); // set two parameters
pstmt.executeUpdate(); // first execution
pstmt.setInt(1,5); pstmt.setString(2,"def"); // set two parameters
pstmt.executeUpdate(); // second execution
```

Parameterized SQL statements are analyzed only once by the DBMS and then executed several times with different parameter values, here (9, "abc") and (5,"def"), by means of `pstmt.executeUpdate()`.

We can now trap preparing statements of the form `pstmt = conn.prepareStatement("UPDATE ...")` in order to obtain the query string. But the execution is later done by `pstmt.executeQuery()` or `pstmt.executeUpdate()` after having set parameter values. Hence, `prepareStatement` itself is not really interesting although the text of SQL operations appears here. Instead, we trap the execution of a prepared statement `pstmt.executeQuery/Update()` by means of another pointcut `jdbcExecutePrepare`. Here, we need access to the `PreparedStatement` object: Using `target(stmt)`, we bind it to a variable `stmt`. Then, `stmt.toString()` can be used later on to determine the SQL statement. That is, pointcuts can be specified in such a way that they expose the context at the matched join point, i.e., object and parameter values can be passed on to our new aspect logic.

Once join points are captured, *advice*s specify weaving rules involving those joint points, such as taking a certain action before or after the join points. The aspect contains advices that now use the above pointcuts to *Notify* and *Request* at adequate joinpoints.

```
after(PreparedStatement stmt) returning(int cnt):
    jdbcExecutePrepare(stmt) {
// cnt yields the number of modified records
    String theTable;
    int newCnt = cnt;
    check type of SQL statement by parsing stmt.toString() and
    determine => theTable
    Integer c = (Integer)changeCounter.get(theTable);
    if (c != null) newCnt = c.intValue() + cnt;
    changeCounter.put(theTable, new Integer(newCnt));
}
```

The clause `after() returning()` is used to add code after the join points specified by `jdbcExecutePrepare`, i.e., after having executed a prepared a statement. The `returning` clause binds a variable `cnt` to the value returned by `stmt.executeUpdate()`, i.e., we can access the number of affected records by `cnt`.

The executed statement is accessible by passing the `PreparedStatement` parameter `stmt` from the pointcut to the advice. The advice obtains the SQL string by `stmt.toString()` and can extract the affected tables. It then either inserts a new entry into the `changeCounters` (if not already existing) or increments the existing value by `cnt`. Analogously, direct executions of statements are handled by defining an advice for `jdbcExecuteUpdate`.

The principle for *Request* is similar. A *before* advice checks all SQL queries to be executed (SELECT, UPDATE, DELETE) whether they are complex:

```
before(String str):jdbcExecuteQuery(str) || jdbcExecuteUpdate(str)
{ determine type and relevant tables from str;
  ArrayList tabList = new ArrayList;
  if (isComplexCondition(str)) {
    for each table occurring in query
    { Integer cnt = (Integer)changeCounter.get(table);
      if (cnt != null &&
          cnt.intValue() >= getThreshold()) { // threshold passed
        tabList.insert(table);
        changeCounter.put(table, new Integer(0));
      }
    }
    computeStatistics(tabList);
  } // else simple condition
}
```

If the query is considered to be complex, for instance, if several tables are joined or if the query contains GROUP-BY-HAVING, the advice extracts the table name(s) from the SQL string *str* to be performed. If a certain threshold for table modifications is passed, then the table is added to a list of tables, and the counter is reset for the table. Finally, the statistics are computed for all the collected tables. That is, a statistics calculation is only performed for a table if a certain amount of records have been modified.

It is important to note that these pointcut definitions and advices do not depend on a concrete application: Aspects work for any JDBC statement in any application – without touching the code of the application. Moreover, we have now reliable and quantified information about staleness: The number of modified records is determined automatically, while a *Notify* could have been forgotten in a manual approach. But some decisions have still to be taken: the threshold and the question what queries are complex. Each application has a different understanding and settings. The same way, *computeStatistics* must be handled for different RDBSs. This is organized as follows: We define our *UpdateStatistics* aspect to be abstract and add abstract methods:

```
public abstract aspect UpdateStatistics {
  public abstract boolean isComplexCondition(String query);
  public abstract int getThreshold();
  public abstract void computeStatistics(List tables);
}
```

Applications can then derive specific aspects that have their own implementation:

```
public aspect UpdateStatistics4Appl extends UpdateStatistics {
  public boolean isComplexCondition(String query) { ... }
  public int getThreshold() { return 5000; }
  public void computeStatistics(List tables) {... for Oracle ... }
}
```

So far, any DELETE, UPDATE and INSERT statement is trapped, no matter whether the surrounding transaction is committed or rolled back. Since only committed transactions have to be taken into account, we catch any rollback and reset the

changeCounter table to the values at the beginning of the transaction by means of

```
public pointcut jdbcRollback :
    call(void java.sql.Connection.rollback());
before jdbcRollback() { resetChangeCounter(); }
```

Unfortunately, the presented solution has one drawback: The environment is not centralized; the statistical data is collected separately for each process. It could now happen that four processes insert 4999 records each (just below a threshold of 5000); then no calculation of statistics would take place for any process although 19996 records have been inserted in total. And if all processes insert the missing record, then all of them would calculate the statistics although only one overall calculation would be enough. In case of our telecommunication scenario, this is no real problem since most important processes are not working in parallel.

Having processes running in parallel, a central environment is required that records the changes for all processes. If all processes run on the same machine (the database server may run on a different one), a simple solution exists: We still use AspectJ for collecting changes as described above, but the number of changed records is passed to shared memory. Whenever an aspect detects a complex query, the shared memory is used to decide whether statistics have to be updated for certain tables.

#### 4 Possible Alternatives

Vendors such as DB2 UDB [AH+04] are currently investigating solutions for automatically keeping statistics up-to-date. However, we here concentrate on solutions that work for any kind of database system, even those that are unable to manage statistics adequately.

We can build a similar environment with database triggers – in principle. AFTER INSERT/UPDATE/DELETE row triggers can be defined for each relevant table. The row triggers increment the counters, however, now using a table COUNTER(tabname,cnt) in the database. In contrast to the AspectJ solution, the performance degradation is much higher, since each inserted, updated, and deleted record is accompanied by trigger activation to collect changes. Due to our investigations, INSERT statements are heavily affected by performance overhead. Statement triggers would be more efficient than row triggers, but cannot be used because it is not possible to compute the number of affected records within the trigger. Anyway, using triggers, we are unable to get the query text in order to recognize what queries are complex. Hence, there is no automatic control on query complexity. Manual work is still necessary.

Well-known design patterns such as Interceptor or Decorator can also be used. However, they require some manual preparation for plugging in the additional behavior. We neglected such an approach because several hundreds lines of code would have to be touched, selected very carefully. Furthermore, our AspectJ approach is configurable, reusable, and has the advantage that it can be switched on and off easily.

Another proposal for staleness could be based on the number of physical blocks used for tables. This number can be computed for tables very cheaply. But on the one hand, the number of records must be derived from the block size – a block might contain 10 large or 1000 small records. On the other hand, the number of blocks is only a rough estimation as a DBMS will usually not release blocks in case of DELETE.

## 5 Results

We tried to assess the effectiveness of our AO approach in a real telecommunication application that had heavy performance problems. A precise performance analysis is not easy to perform. We can certainly compare our solution with the original application, the performance of which is bad and highly non-deterministic depending on how up-to-date DB statistics are. This is useless. It might seem to be reasonable to compete with a perfect, highly optimized system. But the effort will be too high to achieve a perfect performance. e.g., by adding optimizer hints – in fact, this is why we were looking for simpler alternatives. We can also obtain a perfect performance for the existing system by updating statistics permanently for any complex query inside an application (however, without measuring the times for calculating statistics). This is again very difficult to achieve, and such a test would run days.

What we did was to compare our solution with one that calculates the database statistics at the beginning of each application. Well, this is not a perfect comparison as an application is changing data before complex queries run, but it gives us an idea of performance improvement. We used a simple definition of complex query implemented by the parser: A query is complex if two large tables (> 10,000 records), more than three tables, or GROUP BYs are involved. We tested several threshold values in our environment to fine-tune the configuration of parameters.

Finally, the presented approach achieves a performance improvement of 25%. This result is remarkable because we count the times for statistics calculation only in our proposal, but not for the original application. Hence, the delay we introduced is more than compensated by an improved performance. This underlines the necessity for having nearly accurate statistics.

## 6 Conclusions

In this paper, we presented an approach to overcome problems with cost-based query optimization in an elegant way. Cost-based query optimization is often the reason for bad query behavior. More precisely, performance problems arise when database statistics are not up-to-date. We discovered severe performance problems in an existing application in the field of telecommunication. The application holds about 1000 tables with only few millions of records. Unfortunately, there are lots of complex queries running that join tables and use aggregate functions. Those queries are badly optimized and last minutes, but could be performed in milliseconds.

Unfortunately, several RDBMSs do not manage the calculation of statistics appropriately;

the calculation must be explicitly initiated by the user, while the RDBS only keeps statistics so that the optimizer can use it. For applications, it is difficult to find an adequate strategy for determining when to calculate database statistics. Up-to-date statistics are necessary for complex queries, but the computation takes some time. Hence, the calculation should only be done if necessary and only for those tables that have changed.

Our proposal uses the recent technology of aspect-orientation in order to initiate statistics calculation in applications in an adequate manner, relieving the RDBS from this difficult task and working individually for that application. Aspect-oriented languages such as AspectJ help us to add behavior in a modular manner, offering the opportunity to adjust the behavior to the application's needs. The main idea is to extract SQL statements and to determine the number of changed records. Using this information, we can decide when to compute database statistics and on what tables. Owing to AspectJ, this logic can be brought into existing applications by just writing an aspect without explicitly touching existing code. The presented approach provides a reusable asset that can be applied to any JDBC-based application to manage statistics adequately for that application. C++ programs with database accesses in ODBC, embedded SQL, or any Microsoft database interface can be handled the same way by using AspectC++ [AC++], which is similar to AspectJ in concepts, but dedicated to C++.

Our ongoing work is concentrating on using aspect-orientation in other database related topics such performing typical database tasks in applications.

## References

- [AC++] AspectC++: <http://www.aspectc.org>
- [AH+04] Aboulnaga, A., Haas, P., Kandil, M., Lightstone, S., Lohman, G., Markl, V., Popivanov, I., Raman, V.: Automated Statistics Collection in DB2 UDB. Proc. VLDB 2004
- [Ch98] Chaudhuri, S.: An Overview of Query Optimization in Relational Systems. Proceedings of 17th Symp. on Principles of Database Systems, Seattle, Washington, 1998
- [ChS95] Chaudhuri, S.; Shim, K.: An Overview of Cost-Based Optimization of Queries with Aggregates. IEEE DE Bulletin 1995 (Special Issue on Query Processing)
- [EFB01] Elrad, T.; Filman, R.; Bader, A. (eds.): Theme Section on Aspect-Oriented Programming. CACM 44(10), 2001
- [GLS93] Gassner, P.; Lohman, G.; Schiefer, K.: Query Optimization in the IBM DB2 Family. IEEE Data Engineering Bulletin, 1993
- [GHQ95] Gupta, A.; Harinarayan, V.; Quass, D.: Aggregate Query Processing in Data Warehousing Environments. In Proc. of 21st Int. Conf. on VLDB, Zurich 1995
- [GMP97] Gibbons, P.; Matias, Y.; Poosala, V.: Fast Incremental Maintenance of Approximate Histograms. In Proc. of VLDB, Athens 1997
- [Gr93] Graefe, G.: Query Evaluation Techniques for Large Databases. In ACM Computing Surveys 25(2), 1993
- [HC97] Haas, L.; Carey, M.; Livny, M.; Shukla, A.: Seeking the Truth About ad-hoc Join Costs. VLDB Journal 6(3), 1997
- [HN95] Haas, L.; Naughton, J.; Seshadri, S.; Stokes, L.: Sampling-Based Estimation of the Number of Distinct Values of an Attribute. In Proc. of 21st VLDB, Zurich 1995
- [IN92] Ioannidis, Y.; Ng, R.; Shim, K.; Sellis, T.: Parametric Query Optimization. Proc. of 19th Int. VLDB, Vancouver (Canada), 1992

- [Ki01] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W.: An Overview of AspectJ. ECOOP 2001, Springer LNCS 2072
- [KG02] Kienzle, J.; Guerraoui, R.: AOP: Does it Make Sense? The Case of Concurrency and Failures. ECOOP 2002, Springer LNCS 2374
- [La03] Laddad, R.: AspectJ in Action. Manning Publications Greenwich 2003
- [OL90] Ono, K.; Lohman, G.: Managing the Complexity of Join Enumeration in Query Optimization. In Proc. of VLDB, Brisbane 1990
- [PI97] Poosala, V.; Ioannidis, Y.: Estimation without the Attribute Value Independence Assumption. In Proc. of VLDB, Athens 1997
- [RC03] Rashid, A.; Chitchyan, R.: Persistence as an Aspect. In M. Aksit (ed.): 2nd Int. Conf. Aspect-Oriented Software Development Boston, ACM 2003
- [TK02] Tsois, A.; Karayannidis, N.; Sellis, T.; Theodoratos, D.: Cost-Based Optimization of Aggregation Star Queries on Hierarchically Clustered Data Warehouses. Proc of 4th Intl. Workshop on Design and Management of Data Warehouses DMDW 2002, Toronto
- [ViN02] Viglas, S.; Naughton, J.: Rate-Based Query Optimization for Streaming Information Sources. ACM SIGMOD, Madison (Wisconsin) 2002