Exploiting Order Dependencies on Primary Keys for Optimization

Michał Chromiak*, Piotr Wiśniewski*, Krzysztof Stencel **

Institute of Informatics
Maria Curie Skłodowska University
Lublin, Poland*
Faculty of Mathematics and Computer Science
Nicolaus Copernicus University
Toruń, Poland*
Institute of Informatics
University of Warsaw
Warsaw, Poland**

Abstract. Functional dependencies have been used in query optimisation for decades. Moreover, if two domains have a natural ordering of their elements, a functional dependency of them can potentially preserve these orderings, i.e. be a monotonic function. This monotonicity can be exploited by query optimizers. Recently, such monotonic functional dependencies have been termed *order dependencies*.
In this paper we propose a query rewriting method based on order dependencies on primary keys. If an attribute used in the `WHERE` clause has an order dependency on the primary key, such a selection can be replaced by the corresponding condition on the primary key. We have implemented this optimisation method in the integration framework called the *cuboid*. It automates the integration of disparate databases according to the CQS model. Cuboids facilitate injecting dependencies and utilizing them in query optimization.

1 Introduction

Optimisation methods that exploit functional dependencies have already been know for decades. Most of these methods base on the injectivity of the dependency function or its lack. Other properties of this functions are disused. However, if the domain of such a function is ordered and the function itself can preserve this order (i.e. be monotonic). This possibility have been discovered by Jarek Gryz [1–4]. He noted that date columns usually monotonic functions of artificial primary keys. The article [1] proposes a simple method based on this observation. The resulting speedup of query execution ranged from 20% to 50%.

* email: mchromiak@umcs.pl
* email: pikonrad@mat.uni.torun.pl
** email: stencel@mimuw.edu.pl

Following papers [2–4] abstract so-called *order dependencies* and present their proof theory similar to Armstrong's axioms.

Optimisation methods invented by Jarek Gryz and his team are implemented inside IBM DB2 in one of its experimental branches. It allows hoping that soon these methods will be available to the user community. However, users of other DBMSs cannot access them. Therefore, in this paper we struggle to implement similar optimisation mechanisms *outside* of a specific database system. Our experience [5–9] proves that a middleware is a perfect place to do this. It allows e.g. avoiding a dependency of a particular database vendor. In this paper, we use a *cuboid* as such a middleware. [10, 11]

The paper is organized as follows. Section section 2 presents a motivating example that shows optimisation potential values vested in order dependencies. Section section 3 describes the proposed query rewriting algorithm and justifies its semantic correctness. Section section 4 reminds the properties of the cuboid and portrays its potential usage to inject optimisations routines. Section section 5 shows experimental evaluation of the proposed optimisation method. Section section 6 concludes.

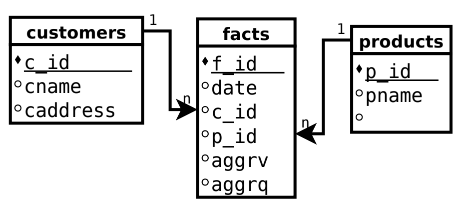
Acknowledgment

We would like to thank Jarek Gryz for his inspiring keynote talk on *Order Dependencies* at the conference BDAS 2014 in Ustroń.

Contribution

The contribution of this paper consists of (1) a query rewriting algorithm that employs order dependencies and (2) an idea to use the cuboid as a layer to inject optimisation algorithms and (3) its positive verification.

Fig. 1. An example schema of a sales database



2 Motivating Example

Assume a simple data warehouse of the schema presented on Figure fig. 1. Consider the query for the sales in a selected period show on Listing 1.1. If the column `date` has no index, this query will require a full scan of the fact table.

Listing 1.1. A query for sales in the indicated period

```
SELECT cname, sum(aggrv), SUM(aggrq)
  FROM facts JOIN customers USING (c_id)
  WHERE date between '2008-12-13' AND '2008-12-15'
  GROUP BY c_id, cname
```

The column `f_id` of the table `facts` is its primary key. Therefore, all other columns of this tables are functionally dependent on it. The column `date` is this a function $d : INT \mapsto DATES$. The implementation of such an artificial key is usually based on a *sequence* generator. Moreover, it is easy to assure that facts on sales from a particular day are recorded *after* all sales from the previous day. Therefore, we can assume that that $d : INT \mapsto DATES$ is non-decreasing. If we assume that:

$$\texttt{xmin} = min\{x; d(x) = \text{'2008-12-13'}\}$$
$$\texttt{xmax} = max\{x; d(x) = \text{'2008-12-15'}\}$$

the query from Listing listing 1.1 is equivalent to the query shown on Listing listing 1.2. We have rewritten the condition of the `WHERE` clause.

Listing 1.2. A rewritten query for sales in the indicated period

```
SELECT cname, sum(aggrv), SUM(aggrq)
  FROM facts JOIN customers USING (c_id)
  WHERE f_id between xmin AND xmax
  GROUP BY c_id, cname
```

The query of Listing 1.2 will be executed using the range index on the primary key provided it is enough selective. Thus, its running time will be significantly shorter than that of the query from Listing listing 1.1. Moreover, the monotonicity of the function d allows efficient computing of `xmin` and `xmax` using the binary search. The experiments we have conducted prove that the overhead caused by this binary search is notably smaller that the time saved by executing the optimized version of the example query. This observations have led us to a rewrite algorithms that implements the idea presented above.

3 Query rewriting algorithm

3.1 Order Dependencies

Assume a table T with its primary key P and remaining attributes $\{A_1, \ldots, A_n\}$. Since P is the primary key of T there exists functions $f_1, \ldots, f_n$ such that each tuple $(p, a_1, ..., a_n)$ of the table T cane be expressed as $(p, f_1(p), \ldots, f_n(p))$. The existence of the functions $f_1, \ldots, f_n$ validate the functional dependencies of the column $A_1, \ldots, A_n$ on the primary key P.

Assume that the domains of the columns P and A_i for a given $i \in \{1, 2, \ldots, n\}$ are linearly ordered sets. The functional dependency between P and A_i will be called an *order Dependencies*, if the function f_i is monotonic. Moreover, an important property of f_i is whether is increasing, non-decreasing, non-increasing or decreasing.

Such dependencies are initially called *monotonic dependencies* [1]. Then, their inventors coin the name *order dependencies*. The motivating example from Section 2 is based on such a dependency between the primary key `f_id` and the column `date`.

3.2 Query rewriting

The goal of the algorithm is to replace range conditions on non-indexed columns to corresponding range search on usually indexed primary key. The algorithm is aware of the schema, functional and order dependencies. Its version presented is this paper is able to rewrite SPJ queries (select-project-joins) and grouping-and-aggregate queries. The input of the algorithm is a query of the form portrayed on Listing1.3:

Listing 1.3. Initial form of the query to be possibly rewritten

```
SELECT ...
  FROM T JOIN T1 ON (T.f1k = T1.pk)
   JOIN T2 ON (T.f2k = T2.pk)
    ....
  WHERE T.Ai BETWEEN  a1 AND a2
  GROUP BY......
```

We also allow `WHERE` clauses with equality and inequality. We than convert them to atomic formulae based on `BETWEEN` using the same value or the data type margin ("infinity") values. The condition `WHERE T.Ai = a` is the converted to `WHERE T.Ai a BETWEEN a`.

In the first step we identify the fact table. We analyze the conditions in the `JOIN ... ON` clauses. The fact table connects other tables by foreign keys while its primary key is not connected by any other foreign key. In a query of the form shown on Listing 1.3 the fact table is denoted by T. If a query contains a string of dependencies foreign-primary key (e.g. in a snowflake schema), the algorithm will also do. However, if it encounters a cycle, it will stop processing and return the original query.

The second step of the algorithm consists in checking whether (1) the `WHERE` clause references a column of the identified fact table and (2) this column has an order dependency of the primary key.

In the third step, if the function f_i is non-decreasing, we will search for values `pmin` and `pmax` such that:

$$\texttt{pmin} = min\{p; f_i(p) = \texttt{a1}\}, \quad \texttt{pmax} = max\{p; f_i(p) = \texttt{a2}\} \tag{1}$$

Analogously, if this function in non-increasing, the algorithm will compute values `pmin` and `pmax` such that:

$$\texttt{pmin} = min\{p; f_i(p) = \texttt{a2}\}, \quad \texttt{pmax} = max\{p; f_i(p) = \texttt{a1}\} \tag{2}$$

Eventually, the algorithm concludes replacing the `WHERE` with:

`WHERE T.P BETWEEN pmin AND pmax`

Since the function f_i is monotonic, the computation of `pmin` and `pmax` can be computed efficiently, e.g. using the binary search.

3.3 Remarks on the Implementation

The algorithm is implemented in a middleware, i.e. outside of the database system. The computation of `pmin` and `pmax` that satisfy conditions (1) and (2) can be done in at least two ways. Both are based on the binary search.

Firstly, we can send a series of queries in the course of the binary search. Its advantage is its inherent simplicity and the lack of any additional database object required. However, it causes numerous communication round trips with the database systems.

Secondly, we can install appropriate stored procedures on the database side. We have decided to implement the binary search exactly this way. When the optimizer on the middleware side is informed on the order dependency between the primary key and the column `date`, it will generate and install two stored functions. One of them shown on Listing 1.4 finds minimal `f_id` for a given date. An analogous function `get_max_fid_by_date(DATE)` that computes maximal `f_id` is also needed.

Listing 1.4. A function that finds the minimal `f_id` for a given date

```
CREATE OR REPLACE FUNCTION get_min_find_by_date (
      DF DATE
 ) RETURNS integer AS $$
DECLARE
     F INTEGER;
     Z INTEGER;
     S INTEGER;
     D DATE;
BEGIN
     SELECT MAX (f_id) INTO Z FROM facts;
     S=1;
     WHILE S<Z LOOP
          S=S*2;
       END LOOP;
     F=S;
     WHILE S>1 LOOP
          S=S/2;
          SELECT date into D from facts where f_id = F - S;
          IF D>= DF THEN
               F=F - S;
            END IF;
```

```
      END LOOP;
    RETURN F;
END;
$$ LANGUAGE plpgsql
```

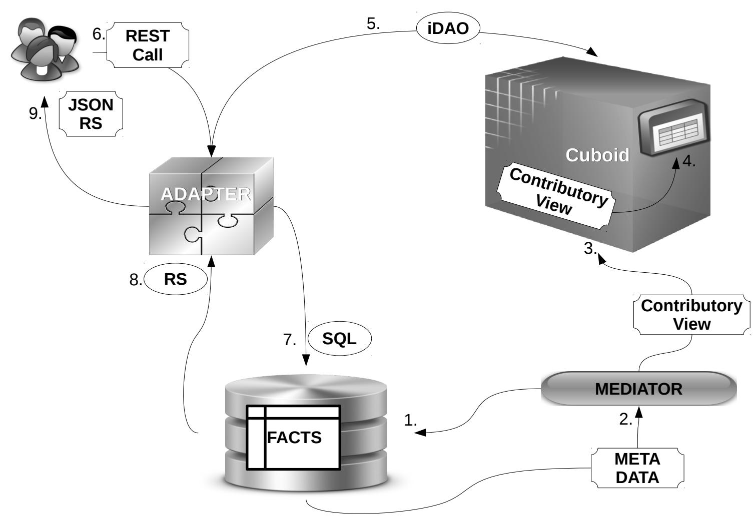
For the sake of readability we removed error handling code from the function `get_min...` These errors may be caused by gaps in the numbering stored in the column `f_id`.

Using this function (and its twin `get_max...`) the optimizer will first issue queries for corresponding margin values of `f_id`. Then, it will put the collected parameters as values of bind variables in the modified query.

4 Cuboid as the linkup data structure

Cuboid is a form of central, master metadata repository. It is responsible for storing contributory meta information about constituent data sources. Each integrated data source must first be a subject of a registration procedure (see fig. 2). To register a data source at the *Cuboid* each data source needs a dedi-

Fig. 2. General architecture for data access



cated mediator to extract from the data source its most informative metadata about schemas, entities and their detailed description. Among those metadata mediator also places adequate queries. Those queries are native queries considering particular data source. Each query is responsible for storing information

about particular part of the schema. The decision about which part of the data is going to be covered with the queries' result sets is for the first time made at mediator configuration, prior to its registration in *Cuboid*. Thus, during mediator initialization (ie. metadata collecting from data source) mediator is aware of the data sets that needs to be covered with queries. The metadata collected from data source can further be modified during the mediator to *Cuboid* chatter. At the *Cuboid*, the metadata, in form of a contributory view provided during mediator registration, is stored and used for building of the global view. The global view is configured and build by designer at the *Cubiod* site. Prepared global view is then made available to the *Adapter* instance in form of *interoperable Data Access Objects (iDAO)*. Those objects are designed to cover each data source contact details and its requested metadata with native queries.
Each time client requests from *Adapter* one of global views[1], *Adapter* reaches for requested global view from Cuboid and unmarshalls out of it native queries, together with contact details to the data source of their origin. Using the contact details *Adapter* sends native queries to original data sources and receives requested result sets. This is happening at the lowest level - ie. JDBC. Now the result sets are being composed together based on global view. When the global view is ready in materialized form, it shall be made available to the client in unified way - ie. in form of REST API.
The entire process involves complex metamodel for collecting and transforming contributory and global metadata. This information has been discussed with details and examples in [11, 10].
The discussed optimization method in form of Order Dependencies (OD) can be easily applied with use of *Cuboid* architecture. Without need to interfere with database optimization engines we will be able to rewrite queries stored at the site of Cuboid.

4.1 Unified data access interface

As depict in fig. 2, client calls for the optimized resource are supposed to be commenced using REST API. The construct of the client REST API includes non optimized and optimized version of the query.

While requesting client will use the REST API to define whether the query response ie. result set, is going to be processed using non optimized version of the query or optimized. We have prepared four implementations for data access layers. Two of them - using JdbcTemplate and SimpleJdbcCall- are Spring Framework based and third is pure JDBC. The final method gets the pmin and pmax hard coded. This is to compare the time of pmin and pmax retrieval and overhead that is brought by each of the three remaining methods.

The REST API is designed as follows. To retrieve unoptimized query answer the request URL should look like this:
`http://localhost:8080/DIAS/rest/dbs/facts/2008-01-01:2008-01-02`

[1] The *Cuboid* can store many arbitrarily customized global views depending on designer requirements.

Now, to request for an optimized query depending on query commuting mechanism the URL would change to:
`http://localhost:8080/DIAS/rest/dbs/facts/2008-01-01:2008-01-02/opti/X`
Where X stands for the query commit method number. The X values has been assigned as follows:

1. Spring simpleJdbcCall (stored functions)
2. Spring JDBCTemplate call statement (stored functions)
3. pure JDBC connection (stored functions)
4. Spring JdbcTemplate with (sub-queries rewrite)
5. hard coded pmin, pmax values

This way we can get the necessary optimization method in simple and straightforward manner. We will present the results in following section.

5 Experiments

Let us first describe the testing environment. The tests has been performed using the following hardware:

CPU	Intel Core i7-3612QM CPU @ 2.10 GHz x 8
RAM	15,6 GiB
Disk	SAMSUNG SSD PM830 2.5” 7mm 512GB
OS	Ubuntu 14.04 LTS
Kernel	3.13.0-30-generic
Arch.	x86_64 GNU/Linux

Table 1: Hardware configuration used for tests.

The procedure was to measure response times for the REST client calls to optimized and unoptimized queries. This means a testing REST client called REST API that has used underneath optimized or non-optimized query for result set retrieval.

The tests has been performed using a the following software:

Java	java version 1.7_60 Java(TM) SE Runtime Environment (build 1.7.0_60-b19) Java HotSpot(TM) 64-Bit Server VM (build 24.60-b09, mixed mode)
REST Testing Client	ApacheBench, Version 2.3
Http Server	Apache Tomcat/6.0.29

Table 2: Software used in testing process.

The test cases assumed two optimization methods. One was to rewrite query with substitution of `WHERE` clause with two stored function results. For comparison reasons, the second case (fifth method) assumed replacing the stored functions with simple sub queries to achieve the same goal as in the firs case. Namely:

Listing 1.5. Simple rewrite with sub-queries

```
SELECT f_id , sum(aggrv) ,
FROM facts
WHERE f_date BETWEEN (select min(f_id) from facts
                      where f_date >= x)
               AND (select max(f_id) from facts
                      where f_date <= y)
GROUP BY c_id;
```

All tested use cases were conducted against the same request parameters and source data. The queried data range was between `2008-01-01` and `2008-01-02`. The result size was 31,546 MB. Each method has been tested 50 times.

Measuring database response times for pmin and pmax was based on Java's `currentTimeMilis()` method from `java.lang.System` [2].

The test results has been placed in table 3.

Activity	**Call Method**									
Document Length [MB]	31.546									
Method Name	**simpleJdbcCall**		**JdbcTemplate**		**JDBC**		**Subquery**		**Hard Coded**	**non-opti**
Stored Functions / Subqueries [ms]	*pmin*	*pmax*	*pmin*	*pmax*	pmin	*pmax*	*pmin*	*pmax*	0	
	13	14	7	6	5	6	7524	15493		
Avg.Time per request [ms]	44.010		29.762		27.196		23038.530		16.431	87681.945

Table 3: Test results for 50 request trial.

The results has clearly shown that rewriting the `WHERE` clause boosts the target query almost 4 times. This is while only modifying the `WHERE` clause with subqueries enabling primary key in role of index. This gives us the idea of how order dependency based query can be effective. Both of the queries do operate on `f_date` column that has not even been indexed. The result would be greatly better if only we would place an index on `f_date` column. The hard coded column values for pmin and pmax are presented to compare the time performance of the query itself without rewriting process.

Three remaining JDBC-based use cases for *(pmin,pmax)* retrieval, are at worst three times slower than the hard coded *(pmin,pmax)* pair.

[2] The detailed discussion for choosing this method has been conducted in [12]

In general we have gained a speed boost form 87.681 seconds to only 0.027 seconds, which reduced the time of result retrieval for approx. 99,96%. Such gain for the discussed use case, is achieved with best - pure JDBC - method, comparing to non optimized query.

6 Conclusions

In this paper we have analyzed so called order dependencies and their optimisation potential when applied at the middleware level. An order dependency is a functional dependency such that its induced function is monotonic with respect to linear orderings of domains. We have proposed an optimisation method that exploits order dependencies on primary keys. We have prepared its proof-of-concept implementation on the middleware level (the cuboid). Middleware has amounted to a feasible place for such optimisations and made such a solution vendor neutral. We have also performed experimental evaluation of this implementation and got promising results.

References

1. Szlichta, J., Godfrey, P., Gryz, J., Ma, W., Pawluk, P., Zuzarte, C.: Queries on dates: fast yet not blind. In Ailamaki, A., Amer-Yahia, S., Patel, J.M., Risch, T., Senellart, P., Stoyanovich, J., eds.: EDBT, ACM (2011) 497–502
2. Szlichta, J., Godfrey, P., Gryz, J.: Fundamentals of order dependencies. PVLDB **5** (2012) 1220–1231
3. Szlichta, J., Godfrey, P., Gryz, J., Zuzarte, C.: Expressiveness and complexity of order dependencies. PVLDB **6** (2013) 1858–1869
4. Szlichta, J., Godfrey, P., Gryz, J., Ma, W., Qiu, W., Zuzarte, C.: Business-intelligence queries with order dependencies in db2. In Amer-Yahia, S., Christophides, V., Kementsietsidis, A., Garofalakis, M.N., Idreos, S., Leroy, V., eds.: EDBT, OpenProceedings.org (2014) 750–761
5. Gawarkiewicz, M., Wiśniewski, P.: Partial aggregation using Hibernate. In Kim, T.H., Adeli, H., Slezak, D., Sandnes, F.E., Song, X., Chung, K.I., Arnett, K.P., eds.: FGIT. Volume 7105 of Lecture Notes in Computer Science., Springer (2011) 90–99
6. Wiśniewski, P., Szumowska, A., Burzańska, M., Boniewicz, A.: Hibernate the recursive queries - defining the recursive queries using Hibernate ORM. In Eder, J., Bieliková, M., Tjoa, A.M., eds.: ADBIS (2). Volume 789 of CEUR Workshop Proceedings., CEUR-WS.org (2011) 190–199
7. Gawarkiewicz, M., Wiśniewski, P., Stencel, K.: Enhanced segment trees in object-relational mapping. In: Proceedings of the 6th Balkan Conference in Informatics. BCI '13, New York, NY, USA, ACM (2013) 122–128
8. Wiśniewski, P., Stencel, K.: Query rewriting based on meta-granular aggregation. In Szczuka, M., Czaja, L., Kacprzak, M., eds.: Proceedings of the 22nd International Workshop on Concurrency, Specification and Programming (CS&P 2013), Białystok, Białystok University of Technology (2013) 457–468

9. Boniewicz, A., Wisniewski, P., Stencel, K.: On materializing paths for faster recursive querying. In Catania, B., Cerquitelli, T., Chiusano, S., Guerrini, G., Kämpf, M., Kemper, A., Novikov, B., Palpanas, T., Pokorný, J., Vakali, A., eds.: ADBIS (2). Volume 241 of Advances in Intelligent Systems and Computing., Springer (2013) 105–112
10. Chromiak, M., Stencel, K.: The linkup data structure for heterogeneous data integration platform. In Kim, T.H., Lee, Y.H., Fang, W.C., eds.: FGIT. Volume 7709 of Lecture Notes in Computer Science., Springer (2012) 263–274
11. Chromiak, M., Stencel, K.: A data model for heterogeneous data integration architecture. In Kozielski, S., Mrozek, D., Kasprowski, P., Malysiak-Mrozek, B., Kostrzewa, D., eds.: BDAS. Volume 424 of Communications in Computer and Information Science., Springer (2014) 547–556
12. Chromiak, M., Lojewski, Z.: Stream security particularities in java. Ann. UMCS, Inf. **8** (2008) 5–13