Query Rewriting Based on Meta-Granular Aggregation

Piotr Wiśniewski[1] and Krzysztof Stencel[2]

[1] Faculty of Mathematics and Computer Science
Nicolaus Copernicus University
Toruń, Poland
pikonrad@mat.uni.torun.pl
[2] Institute of Informatics
The University of Warsaw
Warsaw, Poland
stencel@mimuw.edu.pl

Abstract. Analytical database queries are exceptionally time consuming. Decision support systems employ various execution techniques in order to accelerate such queries and reduce their resource consumption. Probably the most important of them consists in materialization of partial results. However, any introduction of additional derived objects into the database schema increases the cost of software development, since programmers must take care of their usage and synchronization. In this paper we propose novel query rewriting methods that build queries using partial aggregations materialized in additional tables. These methods are based on the concept of meta-granules that represent the information on grouping and used aggregations. Meta-granules have a natural partial order that guides the optimisation process. We also present an experimental evaluation of the proposed rewriting method.

1 Introduction

In the article [1] we presented the idea of materializing partial aggregations in order to accelerate analytical queries. The inherent cost of this idea is attributed to the need of database triggers that keep the materializations up to date in real time. In particular we showed an application programming interface that facilitates defining and using partial aggregations. We designed and implemented appropriate mechanisms that automatically create necessary triggers.

The solutions presented in [1] suffer from a noteworthy deficiency. The application programmer is obliged to cater for additional database objects that store materialized data. He/she not only has to create them, but also has to address them through API methods in order to use them in analytical queries. In particular, it is impossible to use these facilities through queries formulated in HQL, i.e. the standard query language of Hibernate ORM [2].

In this paper we address the abovementioned deficiency. We present an algorithm to rewrite HQL queries so that the usage of materialized aggregations is transparent to application programmers.

This algorithm is based on analyses of the grouping granularity and opportunities to reconstruct necessary data from partial aggregations that are persisted in the database. In order to control the complexity of the space of aggregations that can possibly be materialized, we introduce the notion of a *meta-granule*. It represents a potentially interesting aggregation level. The set of meta-granules is partially ordered. The rewrite method efficiently analyses and traverses the graph of this partial order.

Our solution is based on the idea of materialized views. A recent example of an implementation of such views are *FlexViews* [3] within MySQL based on the results described in [4, 5]. FlexViews rely on applying changes that have been written to the change log.

This article is organized as follows. In Section 2 we recall the idea of partial aggregations and introduce the running example used thorough the paper. We also discuss the integration of the prototype with Hibernate, a major object-relational mapping system. In Section 3 we formalize meta-granules and their partial order. In Section 4 we introduce the query rewrite algorithm that utilizes meta-granules. Section 5 summarizes the results of our experimental evaluation of possible gains triggered by the proposed optimization algorithm. Section 6 concludes.

2 Partial aggregation

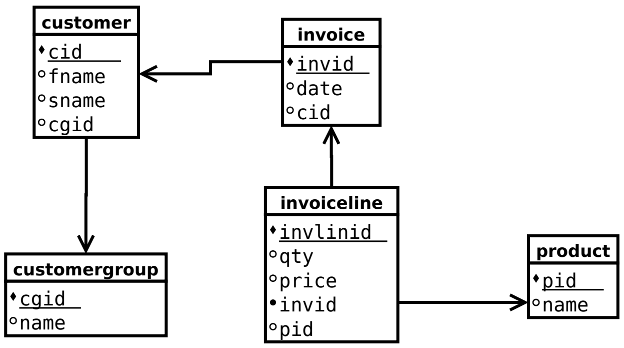

Fig. 1. The original schema of the database on customers and invoices.

Example 1. Let us consider a database schema on customers and their invoices as show on Figure 1. Assume that the company database often has to answer queries that follow the pattern of the SELECT statement presented below.

```
SELECT invoiceline.pid, invoice.date,
       sum(invoiceline.qty)
  FROM invoice JOIN invoiceline USING (invid)
```

```
 GROUP BY invoice.date, invoiceline.pid
HAVING date BETWEEN '2011-07-16' AND '2011-07-22'
```

In order to serve such queries efficiently we extend the database schema from Figure 1 by adding the derived table `dw_invline_value_by_customer_date` as shown on Figure 2. These tables will store partial sums that are needed to quickly answer the query shown above.

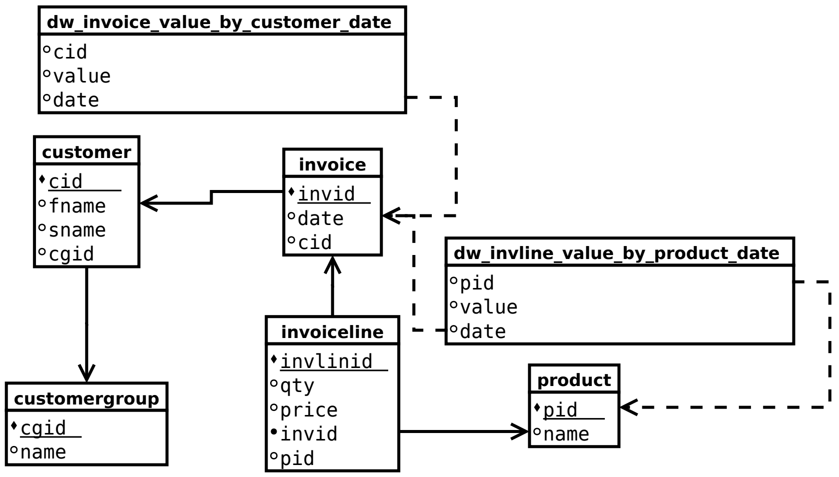

Fig. 2. The schema of the database on customers and invoices extended with derived tables that store materialized aggregations.

In our research we exploit the possibility to hide internals of optimization algorithms in the layers of the object-relational mapping [6]. In our opinion this additional layer of abstraction is a perfect place to put disparate peculiarities of optimisation algorithms. The database server is not the only place to implement query rewriting. Moreover, this ORM option is sometimes the only one, e.g. when the database system is not open-source. It also facilitates writing reusable code that does not depend on the SQL dialect of a particular DBMS. We applied this approach successfully to recursive querying [7] and index selection [8].

Thanks to the generators hidden in the object-relational mapping layer, the extra tables from Figure 2 will be created automatically. For the application programmer it is enough to augment the declaration of the Java entity class `InvoiceLine` with the annotation shown on Listing 1.1.

Listing 1.1. Java class `InvoiceLine` with annotations that cause generation of materialized aggregations

```
@Entity
public class Invoiceline {
     ...
```

```
@DWDim(Dim = "date")
private Invoice invoice;
private Long invlinid;
@DWDim
private Product product;
@DWAgr(function="SUM")
private Integer qty;
...
```

When the materialized aggregations are computed and stored in the database, DBMS can execute the following query using derived storage objects instead of the original user query. The modified query will be served significantly faster since it addresses pre-aggregated data.

```
SELECT pid, date, value
 FROM dw_invline_value_by_customer_date
 WHERE date BETWEEN '2011-07-16' AND '2011-07-22'
```

If we add an appropriate annotation to the entity class `Invoice`, the materialized aggregation `dw_invoice_value_by_customer_date` can be automatically generated. Figure 2 shows this derived table as well. It allows for a notable acceleration of preparation of reports on sales partitioned by dates and customers.

3 Granularity and meta-granules

In this paper *granularity* is the partitioning implied by the grouping clause, while *meta-granules* are schema items that unambiguously define this partition. Basic meta-granules are tables with data we want to summarize, e.g. `invoice_line, unit` is a meta-granule that represents individual rows of the table `invoice_line`. The meta-granule `invoice_line, product, date` stands for the partitioning formalized in our example query. Let us assume that we want to get the total sales for a particular day. For an application programmer it is obvious that instead of base data we can use existing materialized aggregations.

A similar meta-granule `invoice_line, customer, date` describes grouping by the customer and the date of sale. For this meta-granule we created a materialized aggregation as shown of Figure 2. If a user now poses a query for the total sales on a given day, we can choose among two meta-granules that can accelerate his/her query.

Let us introduce a partial order of meta-granules. A meta-granule g_1 is smaller or equal than a meta-granule g_2, if and only if each row in g_2 can be computed by aggregating some rows of g_1.

In the analysis of our running examples we will use the following symbols of meta-granules:

$$g_{il} = \texttt{invoice\_line: unit}$$
$$g_{pd} = \texttt{invoice\_line: product, date}$$

$g_{cd} =$ `invoice_line: customer, date`

$g_d =$ `invoice_line: date`

Then, the following inequalities are satisfied:

$$g_{il} \leq g_{pd} \leq g_d$$

$$g_{il} \leq g_{cd} \leq g_d$$

On the other hand, the meta-granules g_{pd} and g_{cd} are incomparable. Thus, the partial order of meta-granules is not linear.

Let us analyze another example queries that allow identifying other meta-granules.

```
SELECT invoice.date, cg.name
       sum(invoiceline.qty)
  FROM customergroup cg JOIN customer USING(cgid)
    JOIN invoice USING(cid)
    JOIN invoiceline USING (invid)
  GROUP BY invoice.date, cg.name
  HAVING date = '2011-07-16'
```

This query induces the following meta-granule:

$g_{cgm} =$ `invoice_line: customer_group, month(date)`

```
SELECT month(invoice.date), product.pid
       sum(invoiceline.qty)
  FROM invoice JOIN invoiceline USING (invid)
    JOIN product USING (pid)
  GROUP BY month(invoice.date), product.pid
```

This query induces the following meta-granule:

$g_{pm} =$ `invoice_line: product, month(date)`

```
SELECT invid, sum(invoiceline.qty)
  FROM invoice JOIN invoiceline USING (invid)
  GROUP BY invid
  HAVING date = '2011-07-16'
```

This query induces the meta-granule:

$g_i =$ `invoice_line: invoice`

Our extended schema presented on Figure 2 does not contain these meta-granules. If a meta-granule is associated with a materialized aggregation stored in the database, this meta-granule will be called *proper*. Otherwise, the meta-granule is called *virtual*.

Figure 3 shows the partial order of all meta-granules enumerated in presented examples. Virtual meta-granules are depicted as rectangles, while proper met-granules are portrayed as ovals. Observe that each meta-granule is bigger or equal to the proper meta-granule of basic facts, i.e. g_{il}. Data in all meta-granules is derived from g_{il} by some aggregation.

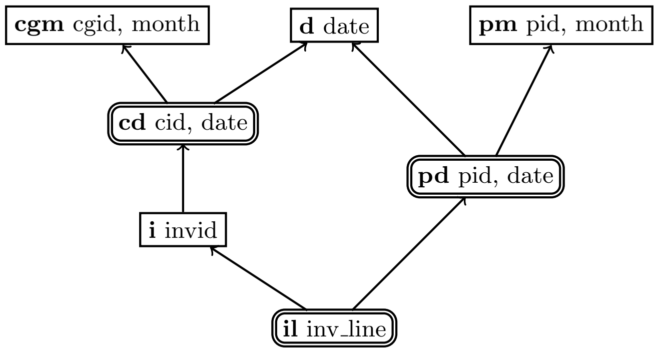

Fig. 3. The partial order of meta-granules

4 The rewriting algorithm

The optimization method based on meta-granules is composed of the following steps. Assume that a query has been posed.

1. We identify the needed meta-granule. We check if the query has internal WHERE clause. Next, we analyze the grouping used and we compute the meta-granule required to compute the answer to the query.
2. If this meta-granule is proper, the query will get rewritten so that it uses the materialization associated with the meta-granule instead of the base fact table.
3. If this meta-granule is virtual, we will find the maximal proper meta-granule not greater than the meta-granule of the query. This maximal meta-granule will be used in the rewriting of the original query.

Since the set of all meta-granules is finite, the set of meta-granules smaller that the given virtual meta-granule always contains maximal elements. This set is never empty, since there exists basic fact meta-granule that is smaller than any mete-granule. However, there can be more than one maximal meta-granule in this set. Let us consider the following example query:

```
SELECT invoice.date, sum(invoiceline.qty)
  FROM invoice JOIN invoiceline USING (invid)
  GROUP BY invoice.date, product.pid
```

This query induces the metagranule:

$$g_d = \texttt{invoice\_line: date}$$

For this virtual meta-granule we have two maximal proper meta-granules. They are g_{cd} and g_{pd}. The usage of any of them means a significant acceleration of the execution of this query.

5 Experimental evaluation

In this Section we show the potential gains of using the optimization algorithm proposed in this paper. We used a computer with Pentium G2120 3.1 GHz (Ivy Brigde), 8 GB RAM. The disk was 120 GB SDD SATA III for the system and Raid 0 on 2x Caviar Black 1 TB 7400rpm for the database storage. We used plain PostgreSQL 9.1 installed on Ubuntu 13.04. No upfront optimization or tuning has been performed. The tested database have the schema shown on Figure 1. The volumes of data are summarized by Table 1.

Table 1. Row counts of tables from the example schema

Table name	Row count
customer	4 999 000
customergroup	50 000
invoice	99 973 000
invoiceline	1 049 614 234
product	9 000

We use three databases instances with different sets of proper meta-granules from Figure 3. The database *plain* contains only the basic meta-granule with invoice lines. The database *med* additionally has proper meta-granules **cd** and **pd**. The database *full* materializes all meta-granules from Figure 3. Table 2 recaps proper meta-granules of all these three databases. It also shows their sizes in gigabytes.

Table 2. Proper meta-granules and volumes of tested database instances

Database name	Proper meta-granules	Volume
plain	`il`	101 GB
med	`cd`, `il`, `pd`	110 GB
full	`cd`, `cgm`, `d`, `i`, `il`, `pd`, `pm`	120 GB

5.1 Rewriting queries

Let us start from a simple query that returns 10 customer groups with biggest sales in March 2006. This query can be formulated as follows assuming the schema from Figure 2.

```
SELECT cgid, sum(price * qty) as sum_val,
       extract(year FROM date) as year,
       extract(month FROM date) as month
```

```
  FROM invline JOIN inv USING (invid)
               JOIN cust USING (cid)
  GROUP BY cgid, year, month
  HAVING extract(year FROM date) = 2006
     AND extract(month FROM date) = 3
  ORDER BY sum_val DESC
  LIMIT 10;
```

Since in the database *plain* no non-trivial proper meta-granule is available, out rewriting algorithm cannot modify this query. When run in this form, it finishes in 1 067.5 seconds.

However, in the database *med* we have the meta-granule **cd** at our disposal. It contains data pre-aggregated by `cid` and `date`. Therefore, our algorithm rewrites the query to use this proper meta-granule:

```
SELECT cgid, sum(sum_val) as cgmsum_val,
       extract(year FROM date) as year,
       extract(month FROM date) as month
  FROM aggr_cd JOIN cust USING (cid)
  GROUP BY cgid, year, month
  HAVING extract(year FROM date) = 2006
     AND extract(month FROM date) = 3
  ORDER BY cgmsum_val DESC
  LIMIT 10;
```

Now the query will run 41.6 seconds. We have accelerated this query 25 times. Although, the database *med* does not contain all possible meta-granules, the running time has been significantly reduced. Of course further increase of efficiency is possible if we have even more proper meta-granules as in the database *full*. In this case, the algorithm will choose using the meta-granule `cgm` to get the following query that runs in half a second.

```
SELECT cgid, sum_val, year, month
  FROM aggr_cgm
  WHERE year = 2006
    AND month = 3
  ORDER BY sum_val DESC
  LIMIT 10
```

In the second example query we ask for 15 best sale days in 2005. This query can be formulated as shown below assuming the schema from Figure 2. Only in this form it can be run against the database *plain* that has no proper meta-granules. It takes 3 475.1 seconds to complete its execution.

```
SELECT date, sum(price * qty) as sum_val
  FROM invline JOIN inv USING (invid)
  GROUP BY date
  ORDER BY sum_val DESC
  LIMIT 15
```

With the database *med* the optimiser has two options. It can use either the meta-granule **cd** or **pd** as presented below. It takes 56.2 sec to completion with **cd** and 19.9 sec with **pd**. Since both queries are plain SQL, the cost model of the underlying database should be used to determine the query to be run. In this case we have two maximal meta-granules to choose.

```
SELECT date, sum(sum_val) as dsum_val
  FROM aggr_cd
  GROUP BY date
  ORDER BY dsum_val DESC
  LIMIT 15;

  SELECT date, sum(sum_val) as dsum_val
  FROM aggr_pd
  GROUP BY date
  ORDER BY dsum_val DESC
  LIMIT 15;
```

When we have all possible proper meta-granules and in the database *full* we can use the meta-granules **d** and run the following query in 41 milliseconds.

```
SELECT date, sum_val
  FROM aggr_d
  ORDER BY sum_val DESC
  LIMIT 15;
```

The third query is to list 10 best selling products together with the sold volume from September to December 2006. In absence of proper meta-granules it can be formulated as follows. In this form for the database *plain* this query runs for 1 074 seconds.

```
SELECT pid, sum(qty) as sum_qty,
       sum (price * qty) as sum_val
  FROM invline JOIN inv USING (invid)
  WHERE extract(year FROM date) = 2006
     AND extract(month FROM date) between 9 and 12
  GROUP BY pid
  ORDER BY sum_val DESC
  LIMIT 10;
```

In the database *med* we can employ the proper meta-granule **pd** and get the following query that completes in 38.8 seconds.

```
SELECT pid, sum(sum_qty) as pmsum_qty,
       sum (sum_val) as pmsum_val
  FROM aggr_pd
  WHERE extract(year FROM date) = 2006
     AND extract(month FROM date) between 9 and 12
  GROUP BY pid
  ORDER BY pmsum_val DESC
  LIMIT 10;
```

The abundant meta-granules of the database *full* allow executing the following query instead. It finishes in 1282 miliseconds.

```
SELECT pid, sum(sum_qty) as pmsum_qty,
       sum (sum_val) as pmsum_val
  FROM aggr_pm
  WHERE year = 2006
     AND month between 9 and 12
  GROUP BY pid
  ORDER BY pmsum_val DESC
  LIMIT 10;
```

The results of the tests are summarized in Table 3. Obviously, the usage appropriate proper meta-granules significantly accelerates the queries. However, even when only limited subset of meta-granules is proper, our rewriting algorithm can notably boost the query execution. This is the case of the database *med*. Although the optimization algorithm did not have the optimal meta-granule, it could successfully employ the meta-granules that were at its disposal.

Table 3. Summary of query execution times for tested database instances

Database	Query 1	Query 2		Query 3
plain	1 067.5 s	3 475.1 s		1 074.2 s
med	41.6 s	56.2 s	19.9 s	38.8 s
full	0.5 s	0.04 s		0.001 s

5.2 Preparing meta-granules

As usual, keeping derived data structures in sync with the base data induces a significant overhead. In this Section we show experiments that assess this overhead. We performed inserting a number of invoices into each database instance. On average each invoice contained 10 lines. We performed two subsequent runs with 10 000 invoices and one run with 15 000 invoices and one with 20 000 invoices. Before each run (but the second with 10 000 invoices) the database management system was shutdown in order to make the database buffer cold initially. Table 4 summarizes the run times. As we can see, avoiding creation of some proper meta-granules (compare *med* with *full*) spares a noteworthy amount of time.

Table 4. Time spent on inserting new invoices and synchronizing proper meta-granules

Invoices	Buffer	*plain*	*med*	*full*
10 000	cold	2m 58.335s	29m 06.670s	37m 56.663s
10 000	hot	3m 03.308s	9m 05.212s	13m 01.924s
15 000	cold	4m 30.261s	20m 59.395s	36m 30.021s
20 000	cold	6m 32.502s	29m 59.064s	44m 38.321s

6 Conclusions

In this paper we presented a novel method to select materialized data in analytical query execution. A fact table can be pre-aggregated for numerous sets of its dimensions. We call this sets of dimensions meta-granules and introduce their partial order. “Bigger” meta-granules are more aggregated, i.e., contain less specific data. Whenever an *ad-hoc* query is posed, the database system can choose using some of the stored meta-granules as a means to accelerate the query. Sometimes, DBMS can find a perfect meta-granule. If such meta-granule does not exists, DBMS will not give up. According to the presented rewriting algorithm, DBMS will use the maximal suitable meta-granule, i.e. the one that is least coarse but still fits the query. Our solution has two benefits. First, the database administrator does not have to create all imaginable materializations. Second, even if some reasonable materialization has been forgotten, the database system can still use exiting imperfect meta-granules to boost the query.

We have also shown results of the experimental evaluation of out method. It proves that even if some ideal meta-granules lack, the database system can still offer satisfactory performance. The experiments also attest that the overhead caused by the need to keep materialized data in sync is acceptable.

References

1. Gawarkiewicz, M., Wiśniewski, P.: Partial aggregation using Hibernate. In: FGIT. Volume 7105 of LNCS. (2011) 90–99
2. O’Neil, E.J.: Object/relational mapping 2008: Hibernate and the Entity Data Model (EDM). In Wang, J.T.L., ed.: SIGMOD Conference, ACM (2008) 1351–1356
3. Flexviews: Incrementally refreshable materialized views for MySQL (2012)
4. Mumick, I.S., Quass, D., Mumick, B.S.: Maintenance of data cubes and summary tables in a warehouse. In Peckham, J., ed.: SIGMOD Conference, ACM Press (1997) 100–111

5. Salem, K., Beyer, K., Lindsay, B., Cochrane, R.: How to roll a join: asynchronous incremental view maintenance. SIGMOD Rec. **29** (2000) 129–140
6. Melnik, S., Adya, A., Bernstein, P.A.: Compiling mappings to bridge applications and databases. ACM Trans. Database Syst. **33** (2008)
7. Szumowska, A., Burzańska, M., Wiśniewski, P., Stencel, K.: Efficient implementation of recursive queries in major object relational mapping systems. In: FGIT. (2011) 78–89
8. Boniewicz, A., Gawarkiewicz, M., Wiśniewski, P.: Automatic selection of functional indexes for object relational mappings system. International Journal of Software Engineering and Its Applications **7** (2013)