Granular Indices for HQL Analytic Queries

Michał Gawarkiewicz, Piotr Wiśniewski, and Krzysztof Stencel

Faculty of Mathematics and Computer Science
Nicolaus Copernicus University
Toruń, Poland
{garfi,pikonrad,stencel}@mat.umk.pl

Abstract. Database management systems use numerous optimization techniques to accelerate complex analytical queries. Such queries have to scan enormous amounts of records. The usual technique to reduce their run-time is the materialization of partial aggregates of base data. In previous papers we have proposed the concept of metagranules, i.e. partially ordered aggregations of the fact table. When a query is posed, the actual aggregation level will be determined and the smallest fit metagranule (materialized aggregation) will be used instead of the fact table. In this paper we extend that idea with metagranular indices, i.e. indices on metagranules. Assume a user issuing an aggregate query to a fact table with a selective `HAVING` or small `LIMIT-ORDER BY` clause. The database engine can not only identify the best metagranule but it can also use the index on that metagranule in order not to scan its full content. In this paper we present the proposed optimization method based on metagranular indices. We also describe its proof-of-concept prototype implementation. Finally, we report the results of performance experiments on database instances up to 350GiB.

Keywords: analityc queries, ORM, databases.

1 Introduction

Processing analytical queries is time-consuming, since they require scanning countless records. A database admin can avoid such enormous scans by materializing properly pre-aggregated data. On the other hand, he/she must also take into account the space occupied by such redundant aggregates and the overhead on updates they bring about. Therefore, an administrator has to find an equilibrium between gains (faster queries) and costs (space and processing overhead). This activity may be supported or even performed by automated tools.

The problem how to choose a proper set of indices for an application is an interesting research topic [3]. Analyses of database workloads lead to a number of index sets that can speed up query processing. The maintenance cost of each of these sets is then computed and compared with the profits. The set with biggest difference between profits and costs is suggested to a database admin [6,10,13,5,4]. A number of index advisors has been developed in commercial databases. Since the database workload changes over time, advisors have to be

S. Kozielski et al. (Eds.): BDAS 2014, CCIS 424, pp. 30–39, 2014.

rerun over and over again. Therefore, online index advisors [2] and benchmarks to compare them have emerged [14]. Further research on the choice of indices led to adaptive indices, e.g. *database cracking* [11] and *adaptive merging* [9]. Since both methods prove useful, a hybrid approach has also been proposed [12]. There is also a benchmark to compare adaptive indices [8].

In our research we focus on the object-relational mapping middleware (ORM) and its inherent optimisation potential. We showed that ORM could optimize analytic queries by materialization of partial aggregates [7]. We also described a programming interface to define aggregations worth materializing (called metagranules) and a query rewriting that facilitates using them. In this paper we extend that approach by considering usage of indices on metagranules. Assume a query rewritten so that it uses materialized aggregates instead of base data. If this query contains a `HAVING` or `ORDER-BY` clause (preferably combined with `LIMIT`), an index on the used metagranule can further accelerate the query. Queries are analysed using a method similar to the one presented in [1]. The proposed method collects queries from an application, and then examines them to find indices that will potentially accelerate the application. In our proof-of-concept prototype, we use Hibernate ORM. Thus, the queries considered are HQL (Hibernate QL) queries. In this paper, we report results of experimental evaluation of this prototype on database instances up to 350 GiB. The results attest that the proposed method can significantly increase the efficiency of applications that use various ORMs. This paper makes the following contributions:

- we propose a novel method to advise and automatically generate indices on metagranules;
- we describe a query rewriting method that aids using these indices;
- we show our proof-of-concept prototype implementation of these ideas;
- we show performance evaluation of this prototype using database instances of the size up to 350GiB.

The paper is organized as follows. In section 2 we motivate the idea of granular indices. Section 3 reminds our approach to materializing partial aggregates in ORMs. In section 4 we describe the concept of metagranules and discuss the cost aspect of storing them. Section 5 presents the algorithm that analyses queries collected from an application. Section 6 reports the result of experimental evaluation of the proposed method. Section 7 concludes.

2 Motivation

Assume a business analysis application with a database of the schema as presented on Figure 1. Its user poses the query shown on Listing 1.1. In the absence of auxiliary data structures such query has to scan all rows of the fact table (`invline`). However, if the aggregation of the expression `(inl.price * inl.qty)` by customer's id is materialized, the query can be answered significantly faster. Further acceleration can be achieved if we create an index on the summed column of such materialized aggregate. In such a case, the query engine

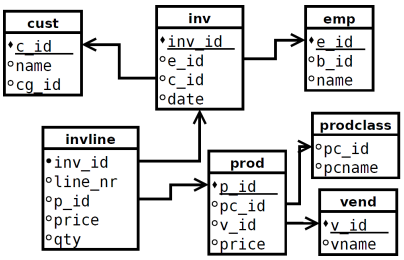

Fig. 1. The schema of a sample business database

Listing 1.1. The query to find twenty best customers

```
SELECT cust.cid, SUM(inl.price * inl.qty)
  FROM cust JOIN inv USING (c_id)
       JOIN inl USING (inv_id)
  GROUP BY cust.c_id
  ORDER BY SUM(inl.price * inl.qty) DESC
  LIMIT 20;
```

simply collects twenty tail entries from this index (provided it is stored in the ascending order).

The algorithms presented in this paper can *automatically* detect such an optimisation opportunity and suggest creating the corresponding materialized aggregate and its index. However, one has to be aware that maintaining them is costly. Therefore, we also report the results of an experimental analysis of costs and profits induced by these redundant data structures (see section 6).

3 Partial Aggregation

In this paper we focus on applications that use an object-relational mapping system to store their persistent data. Since contemporary ORMs have relatively limited functionality, an application programmer that wants to code analytic processing has two equally terrible options. He/she can accept notably low performance or bypass ORM mechanisms by directly addressing the database with SQL queries [7]. We proposed an extension to Hibernate that allows materializing aggregates without breaking the architecture of an application. Listing 1.2 shows an example class augmented with annotations `@DWDim` and `@DWAggr` that indicate aggregates to be materialized. Our extension interprets such annotations and (1) creates a table that stores `SUM(quantity*price)` grouped by `inv_id, date, c_id`, (2) augments the class `Invoice` with methods to query this table and (3) adds triggers that will synchronize this table with the base data.

Listing 1.2. The class `Invoice` with annotations on metagranules

```
@Entity
```

Listing 1.3. The query for twenty best customers using a finer aggregation

```
SELECT cust.c_id, SUM(sum_qty_x_price)
  FROM mg_inv
  GROUP BY cust.c_id
  ORDER BY SUM(sum_qty_x_price) DESC
  LIMIT 20;
```

```
@Granule(Dim = "id, date, customer"
         Agr = "Sum(invLines.quantity*invLines.price)")
@Granule(Dim = "date,....")
public class Invoice {
    @DWDim
    private Long id;
    @DWDim
    private Date date;
    @DWDim
    private Customer customer;
    @DWAggr(function="SUM(quantity*price)")
    private List<InvoiceLine> invLines;
}
```

Usually more than one subset of dimensions is used to aggregate data. In order to pass the set of all interesting aggregations we can use the annotation `@Granule` [15] as shown on Listing 1.2. This annotation indicates that the table `mg_inv` with columns `inv_id`, `date`, `c_id`, `sum_qty_x_price` is to be created. The first three columns are dimensions, while the fourth column is `SUM(quantity*price)` over these three dimensions.

4 Metagranules

Annotations presented in section 3 may impose a significant overhead on the database. Storing and maintaining numerous materialized aggregates requires space and time to synchronize data. However, we can observe that some analytic queries can be rewritten so that they used more finely aggregated data. The query from Listing 1.1 can be executed as the query from Listing 1.3. Although the original query aggregates over the customer id only, it can benefit from using finer aggregation over invoice id, date and customer id.

This query will not run in a fraction of seconds. For a 100 GiB database the query should finish in less than one minute. This is probably acceptable. The idea of materializing only a subset of desired aggregates has been presented [15]. Grouping levels called *metagranules* constitute a partial order. Figure 2 shows an example of such order. It contains a possible set of metagranules for the database schema from Figure 1.

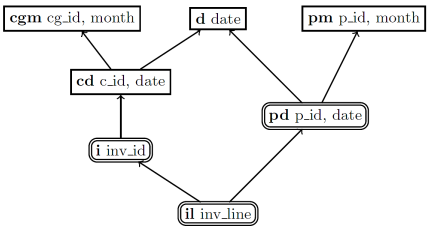

Fig. 2. The partial order of metagranules

Metagranules represent the aggregates used by the application. Some of them are chosen to be actually materialized. We call them *proper metagranules*. In Figure 2 their symbols have double border.

In the article [15] we presented methods to rewrite queries so that they use the most fit proper metagranule, i.e. the maximal metagranule smaller or equal to the desired metagranule. A smaller metagranule contains more records. Thus the query based on a smaller metagranule will finish later. For some metagranules there could be more than one metagranule that satisfies the abovementioned conditions. The metagranule `d` has two such proper metagranules: `i` and `pd`. Eventually, the algorithm chooses the one with smaller number of records. In [15] we performed experiments on a database instance of size 100 GiB. They confirmed the validity of this approach. This idea can be converted into an algorithm as presented in section 5.

The choice of the best set of proper metagranules constitutes another interesting problem. The more metagranules are proper the faster are the queries. On the other hand each proper metagranule induces a noteworthy cost of overhead, since it occupies space and slows down updates. In section 6 we show results of tests against database instances of the sizes 100GB and 350GB. These results reveal the impact of the introduction of new proper metagranules.

If a metagranule is proper, it can also have indices. Therefore, we have adapted the algorithms presented in [1] so that they can be used to suggest metagranular indices. Modified algorithms are discussed in section 5.

5 Processing HQL Queries

In this section we present the algorithms used in the proposed method. We start from the analysis of HQL queries to collect all metagranules possibly worth materializing. Assume that we have already collected the potential database workload, i.e. a set of queries from an application code. Initially the set of metagranules is empty. Then for each query the algorithm performs the following steps:

1. Build the abstract syntax tree (AST) for the query.
2. Locate the `WHERE` node and if it exists, find all database columns in its subtree, but not in aggregate functions.
3. Add all but aggregated columns to the set M of metagranule dimensions.

Table 1. Row counts in the tested database instances

Table	Small DB	Medium DB	Big DB
customer	360 000	600 000	1 200 000
invoice	59 995 970	199 948 701	601 628 100
invoiceline	629 949 439	2 099 416 306	6 316 455 713
product	12 000	24 000	36 000
product day aggr.	29 404 062	59 387 618	88 920 000

4. Find all non-aggregated columns in the subtrees of `GROUP`, `ORDER` and `HAVING` add them to the set M.
5. Find the aggregating function F in the AST.
6. Add M as the dimensions and F as the aggregate of the new metagranule.

Now we focus on the identification of possible metagranular indices for a query. The corresponding algorithm performs the following steps:

1. Find the metagranule for the query.
2. Build the abstract syntax tree (AST) of the query.
3. For each of the `WHERE`, `ORDER` and `HAVING` nodes in the AST, find all references to database columns that are also dimensions of the metagranule.
4. Add them as the columns of a potential metagranular index.

Finally, let us consider a query that can be potentially rewritten to use metagranules and metagranular indices. The corresponding algorithm performs the following steps:

1. Identify the metagranule for this query. If there is none, stop.
2. Build the abstract syntax tree (AST) of the query.
3. Remove from `GROUP` all columns that are dimensions of the metagranule.
4. If the `GROUP` node has no children, remove it.
5. Replace columns and aggregates that are members of the metagranule with columns from the table that stores the aggregated data of the metagranule.
6. Remove tables from the `FROM` node that are no longer referenced.
7. Add the table that stores the data of the metagranule to the `FROM` node.

6 Performance

We used a computer with Intel i5 3570 (ivy bridge) and 8GiB RAM. The storage was Raid 0 over 4xBlack Caviar 1TB controlled by Adaptec 2405. The operating system was Ubuntu 13.10. We used PostgreSQL 9.1. We tested against three database instances: small (base data of size 34 GiB), medium (130 GiB) and big (345 GiB). Each of them was tested in three variants. The variant *plain* contained just the base data. The variant *aggr* stored aggregated data on sales in metagranule tables `mg_inv` (aggregated by invoice) and `mg_pd` (aggregated by day). The instances in the variant *plain* have corresponding sizes 41 GiB, 134 GiB and 391 GiB. In the *index* variant, the medium and big databases have an

Table 2. The execution times of the queries for five best invoices on a given day

Variant	Medium DB	Big DB
plain	820 s	6 410 s
aggr, based on mg_inv	26 s	84 s
index (date) on mg_inv	0.4 s	0.8 s
index (date,sum_qty_x_price) on mg_inv	0.3 s	0.3 s

index on `date` and `sum_price_x_qty`. These indices increased the size of these databases to 145GiB and 418 GiB. Below we report and discuss the execution times of three analytic queries. For the sake of readability we present them in SQL, since there can be interested readers who are not familiar with HQL.

Listing 1.4. A query for five the best invoices on a given day

```
SELECT name, SUM(qty * price) AS sum_qty_x_price
  FROM invline JOIN inv USING (inv_id)
    JOIN cust USING (c_id)
  WHERE date = '2013.11.10'
  GROUP BY name, inv_id
  ORDER BY sum_qty_x_price DESC LIMIT 5;
```

Listing 1.5. The optimized query for five the best invoices on a given day

```
SELECT date, name, sum_qty_x_price
  FROM mg_inv JOIN cust USING (c_id)
  WHERE date = '2013.11.10'
  ORDER BY sum_qty_x_price DESC LIMIT 5;
```

Listing 1.6. A query for total sales for a given month sliced into days

```
SELECT date, SUM(qty * price)
  FROM invline JOIN inv USING(inv_id)
  WHERE EXTRACT(year FROM date) = 2013 AND
    EXTRACT(month FROM date) = 8
  GROUP BY date;
```

The first query finds five biggest invoices on a given day. Listing 1.4 shows its initial (user submitted) form. According to the methods discussed in this paper, this query gets rewritten to the form shown on Listing 1.5. The index suggesting algorithm presented in section 5 picked the indices `(date)` and `(date,sum_qty_x_price)` on the metagranule table `mg_inv`. Tab. 2 shows execution time of this query for various database instances and their variants. We can see that using a materialized aggregation significantly accelerates the query. If the system is severely loaded with such queries, it will be worth building the metagranular index.

The second query finds total sales for each day of a given month as shown on Listing 1.6. The result of this query can be computed from stored aggregates by

Table 3. Execution times of the query for total sales for a given month sliced into days

Variant	Small DB	Medium DB	Big DB
plain	365 s	2 494 s	> 2h
aggr, based on mg_inv	14 s	49 s	147 s
aggr, based on mg_pd	25 s	18 s	27 s

invoices or by products on particular days. Tab. 3 presents the execution times of this query for various database instances and their variants without indices.

Listing 1.7. A query for eight best vendors of a given month

```
SELECT vname, SUM(qty * il.price) AS sum_qty_x_price
  FROM vend JOIN prod USING(v_id)
    JOIN invline il USING (p_id)
    JOIN inv USING(inv_id)
  WHERE EXTRACT(year FROM date) = 2013 AND
    EXTRACT(month FROM date) = 8
  GROUP BY v_id, vname
  ORDER BY sum_qty_x_price DESC LIMIT 8;
```

Listing 1.8. The rewritten query for eight best vendors of a given month

```
SELECT vname, SUM(sum_qty_x_price) as sum_qty_x_price
  FROM vend JOIN prod p USING(v_id)
    JOIN mg_pd USING(p_id)
  WHERE EXTRACT(year FROM date) = 2013 AND
    EXTRACT(month FROM date) = 8
  GROUP BY p.v_id, vname
  ORDER BY suma DESC LIMIT 8;
```

Table 4. The execution times of the query for eight best vendors of a given month

Variant	Small DB	Medium DB	Big DB
plain	365s	1 521s	> 1h
aggr, based on mg_pd	10s	49s	31s

The third example query lists eight best vendors of a given month, i.e. vendors whose products had biggest sales. Listing 1.7 shows this query in the original form. Listing 1.8 presents the rewritten version that uses the materialized aggregation `mg_pd`. Tab. 4 reports executions times of these two version in various database instances.

It is obvious that one cannot store all desired aggregations. However, our experiments show that even a reasonably small subset of materialized aggregations can make the performance of big analytic queries acceptable.

Table 5. The impact of the presented methods in case of the big database

	Size		Insertion of 100 000 invoices		The query from Listing 6	
Version	GiB	Ratio	Time	Ratio	Time	Ratio
plain	345	100%	4.63s	100%	6 410s	100%
aggr	391	113%	556.97s	12 029%	84s	1.31%
index	418	121%	858.59s	(154%) 18 544%	0.3s	(0.36%) 0.0046%

Table 6. Time necessary to insert records on 100 000 invoices

Variant	Medium DB	Big DB
plain	5, 39 s	4, 63 s
aggr	326, 82 s	556, 97 s
index	598, 31 s	858, 59 s

We have also assessed the overhead imposed by proper metagranules. We have analysed the time needed to insert records on 100 000 invoices into three analysed database variants of three different sizes. The results are shown in Tab. 6. For a database variant with stored aggregations, appropriate triggers synchronize derived data. This causes also corresponding updates in granular indices, if they exist.

Tab. 5 shows how the presented methods impact the size of the database and the execution times of 100 000 insertions of invoices and the query from Listing 1.4. This summary contains data on the big database instance that stores 6 billions records in the table `invline`. In parentheses there are ratios of corresponding values from the last two rows.

7 Conclusions

In this paper we extend our previous research on metagranules, i.e. a partial order of possible materialized aggregates that accelerate analytic queries. We show algorithms that find possibilities to build indices on stored aggregations to further improve the efficiency of querying. Performed experimental evaluation proves that using metagranules and their indices can significantly improve the efficiency of an application. However, the overhead imposed by them is noteworthy. Therefore, one cannot materialize all desired aggregations. A quantitave model to balance the cost and profits of metagranules is needed to choose the optimal set of metagranules and indices. The development, tuning and verification of such model is a topic for our further research.

References

1. 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, 189–195 (2013)

2. Bruno, N., Chaudhuri, S.: An online approach to physical design tuning. In: ICDE, pp. 826–835 (2007)
3. Chaudhuri, S., Narasayya, V.R.: An efficient cost-driven index selection tool for Microsoft SQL Server. In: Proceedings of the 23rd International Conference on Very Large Data Bases, VLDB 1997, pp. 146–155. Morgan Kaufmann Publishers Inc., San Francisco (1997), http://dl.acm.org/citation.cfm?id=645923.673646
4. Choenni, S., Blanken, H., Chang, T.: Index selection in relational databases. In: Proc. International Conference on Computing and Information, pp. 491–496 (1993)
5. Choenni, S., Blanken, H.M., Chang, T.: On the automation of physical database design. In: Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing: States of the Art and Practice, SAC 1993, pp. 358–367. ACM, New York (1993), http://doi.acm.org/10.1145/162754.162932
6. Finkelstein, S., Schkolnick, M., Tiberio, P.: Physical database design for relational databases. ACM Trans. Database Syst. 13(1), 91–128 (1988), http://doi.acm.org/10.1145/42201.42205
7. 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 2011. LNCS, vol. 7105, pp. 90–99. Springer, Heidelberg (2011)
8. Graefe, G., Idreos, S., Kuno, H., Manegold, S.: Benchmarking adaptive indexing. In: Nambiar, R., Poess, M. (eds.) TPCTC 2010. LNCS, vol. 6417, pp. 169–184. Springer, Heidelberg (2011), http://dl.acm.org/citation.cfm?id=1946050.1946063
9. Graefe, G., Kuno, H.: Self-selecting, self-tuning, incrementally optimized indexes. In: Proceedings of the 13th International Conference on Extending Database Technology, EDBT 2010, pp. 371–381. ACM, New York (2010), http://doi.acm.org/10.1145/1739041.1739087
10. Hammer, M., Chan, A.: Index selection in a self-adaptive data base management system. In: Proceedings of the 1976 ACM SIGMOD International Conference on Management of Data, SIGMOD 1976, pp. 1–8. ACM, New York (1976), http://dl.acm.org/citation.cfm?id=509383.509385
11. Idreos, S., Kersten, M.L., Manegold, S.: Database cracking. In: CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, pp. 68–78 (2007) (Online Proceedings)
12. Idreos, S., Manegold, S., Kuno, H., Graefe, G.: Merging what's cracked, cracking what's merged: adaptive indexing in main-memory column-stores. Proc. VLDB Endow. 4(9), 586–597 (2011), http://dl.acm.org/citation.cfm?id=2002938.2002944
13. Rozen, S., Shasha, D.: A framework for automating physical database design. In: Proceedings of the 17th International Conference on Very Large Data Bases, VLDB 1991, pp. 401–411. Morgan Kaufmann Publishers Inc., San Francisco (1991), http://dl.acm.org/citation.cfm?id=645917.758359
14. Schnaitter, K., Polyzotis, N.: A benchmark for online index selection. In: Proceedings of the 2009 IEEE International Conference on Data Engineering, ICDE 2009, pp. 1701–1708. IEEE Computer Society, Washington, DC (2009), http://dx.doi.org/10.1109/ICDE.2009.166
15. Winiewski, P., Stencel, K.: Query rewriting based on meta-granular aggregation, pp. 457–468, http://csp2013.mimuw.edu.pl/proceedings/PDF/paper-40.pdf