Piggyback Profiling: Enhancing Query Results with Metadata

Claudia Exeler[1], Maria Graber[3], Tino Junge[1], Stefan Ramson[2], Cathleen Ramson[3], Fabian Tschirschnitz[1], and Felix Naumann[2]

[1] Hasso Plattner Institute, University of Potsdam, Germany
firstname.lastname@student.hpi.de
[2] Hasso Plattner Institute, University of Potsdam, Germany
firstname.lastname@hpi.de
[3] Hasso Plattner Institute, University of Potsdam, Germany
firstname.lastname@hpi-alumni.de

Abstract. SQL-based data exploration is tedious. Any given query might be followed up by another query simply to count the number of distinct values of an interesting column or to find out its range of values. Besides the actual query output, each result also embodies various metadata, which are not visible to the user and not (yet) determined by the DBMS. These metadata can be useful to understand the data, assess its quality, or spark interesting insights.
Our approach piggybacks metadata calculation on the usual query processing with minimal overhead, by making use of specific properties of the query plan nodes. We describe our extension of an RDBMS and show that its runtime overhead is usually less than 10%.

1 Statistics for Exploratory Queries

Imagine Alice – she works as a database engineer at a major music label and wants to integrate the Musicbrainz data dump [8] into an existing music knowledge base. The dataset, which consists of various information about music releases, is rather large (5 GB) and consists of 119 different relations. After creating the tables with a provided script and loading the data into a DBMS, she now wants to get a rough overview of the most important relations. For that reason she starts her exploration by executing the following simple query:

```
SELECT *
FROM   recording rec;
```

Apart from the query result itself, a typical DBMS-client would display only the count of fetched rows as metadata. Alice discovers the attribute rec.length, is now interested in the longest song, and issues a second exploratory query with `MAX(rec.length)` aggregation. We suggest to automatically display such and more metadata of any query result, including the minimum and maximum value, number of distinct values, etc. of a column.

Next, Alice finds out that there are recordings of length −1. Probably, such recording lengths are incorrect and a surprising edge case that she might have overlooked. Our system responds to a query with both data and metadata – a feat not possible with regular SQL interfaces. To further investigate the different concepts of the track and recording table, she joins those two and aggregates a numeric difference of their two length attributes using the query below. She obtains the result shown in Figure 1.

```
SELECT rec.id, rec.name,
       rec.length - tr.length as ldiff,
       tr.id, tr.name, tr.position
FROM   recording rec, track tr
WHERE  rec.id = tr.recording;
```

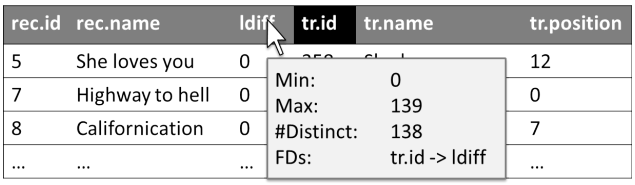

Fig. 1. Output of the second query

Figure 1 shows a possible result-rendering that includes metadata annotations. For instance, the dark column heading of tr.id indicates that it is a key candidate within the result set. So Alice can deduce cardinalities: a recording might have multiple manifestations in terms of tracks, but not vice versa (rec.id is not shaded dark). Further, the first result rows might mislead her concerning the computed length difference. But from the metadata in the tooltip she can conclude that indeed there are length differences of up to 139ms.

Our solution enables the DBMS user to explore the data with fewer queries, especially the typical aggregation-style queries on top of the original queries. Further, it encourages the user to investigate deeper and to ask the system questions that otherwise would not have arisen. The described feedback loop between the semantics in the mind of the developer, the queries, and the metadata about their results is also a form of query debugging. Developers often anticipate certain properties and can now easily verify them without follow-up queries. We thus contribute for example to the "assisted-interaction" use-case outlined in [5] to support developers.

The basic idea of our approach is to analyze the data as they pass through the execution plan, piggybacking the query execution plan nodes. Using the DBMS base statistics and our knowledge of the individual operator semantics we incur only little overhead. Our contributions are a *systematic analysis* of the influence of the different SQL operators on various metadata (Section 3), a

prototypical implementation as an *extension of PostgreSQL* (Section 4), and a *runtime evaluation* on different datasets (Section 5).

2 Related Work

As Morton et al. stated, there is a rising number of users, such as data analysts and data journalists, who deal with large data sets [7]. This fact challenges data analysis systems to provide fast insights into large data sets. Typically, such systems make use of visualization techniques [4], which can provide interesting insights through the recognition of patterns. These techniques heavily rely on the visual data mining capabilities of the human mind. In contrast, our approach aims to generate insights by presenting the available metadata of the query in question.

Database systems typically maintain a set of base statistics for optimization purposes. These base statistics are typically used to optimize query execution trees [6]. We benefit from these statistics by using them as a basis for gathering metadata on the query result.

To keep the runtime overhead small, we collect metadata during query execution as all necessary data is processed anyway. Related approaches of monitoring metadata during query execution have been proposed. However, these approaches focus on a different goal. For instance, Carey and Kossmann aim to place the `Limit` operator as early/low as possible in the execution tree to reduce the runtime of the overall query execution [1]. To this end, the influence of common operators on the number of tuples is monitored during query execution. The Leo system monitors the number of tuples during query execution measuring the actual selectivity of predicates [9]. The measured data are not presented to the user but rather fed back into the optimizer. The introduced feedback loop allows the optimizer to learn from its past mistakes. In contrast to both approaches, which optimize query execution plans, our approach focuses on metadata that are interesting to the user.

3 Piggybacking Query Processing

We investigate the collection of interesting metadata with as little extra effort as possible during regular database query processing. The metadata we consider are unary functional dependencies (FDs) and unique column combinations (UCCs), as well as single column metadata, namely the number of distinct values (#Distinct), minimum (Min), maximum (Max), and most frequent value (MFV) of each column. The collection of further metadata, such as average, number of null values, etc., is analogous.

3.1 Information sources for metadata

Metadata about query results can come from three sources: base statistics, calculations during query processing, and finally the SQL statement itself. Schematic

Table 1. Influence of common operators on different types of metadata

	#Distinct	#Distinct in other columns	Min	Max	Min/Max in other columns	MFV	MFV in other columns	FDs	UCCs
SORT (ASC)	prev. val.	prev. val.	prev. val. (found in first row)	prev. val. (found in last row)	prev. val.	prev. val.	prev. val.	no change	no change
LIMIT y	MIN(y, prev.val.) as boundary	n/a	previous as boundary	previous as boundary	n/a	unknown	n/a	possibly additional	possibly additional
SELECTION	previous as boundary	previous as boundary	previous as boundary	previous as boundary	previous as boundary	prev. val. unless filtered out	unknown	possibly additional	possibly additional
A = y (const.)	1	— " —	y	y	— " —	y	— " —	*any column* $\rightarrow A$	A not part of any minimal UCC
A >= y (const.)	previous as boundary	— " —	MAX(y, prev. val.) as boundary	prev. val.	— " —	prev. val. unless filtered out	— " —	possibly additional	possibly additional
A = B	MIN(dist(A), dist(B))	— " —	MAX(min(A), min(B))	MIN(max(A), max(B))	— " —	— " —	— " —	— " —	both unique, if A or B unique before
A >= B	previous as boundary	— " —	MAX(min(A), min(B))	previous as boundary	— " —	— " —	— " —	— " —	possibly additional
DISTINCT/ GROUP	prev. val.	n/a	prev. val.	prev. val.	n/a	unknown	n/a	no change	group columns are UCC
B = AGGREGATION(A)	unknown	n/a	unknown	unknown	n/a	unknown	n/a	*Group Columns* $\rightarrow B$; possibly addi- tional	possibly additional
B = MIN(A)	previous as boundary	n/a	prev. val.	previous as boundary	n/a	— " —	n/a	$B \rightarrow X$ if $A \rightarrow X$; $X \rightarrow B$ if $X \rightarrow A$	unique if A was unique
B = AVG(A)	unknown	n/a	previous as boundary	— " —	n/a	— " —	n/a	$X \rightarrow B$ if $X \rightarrow A$	possibly additional
JOINS R.A = S.A	MIN(dist(R.A), dist(S.A)) as boundary	previous as boundary	MAX(min(R.A), min(S.A)) as boundary	MIN(max(R.A), max(S.A)) as boundary	previous as boundary	prev. val. if same in R and S; else unknown	unknown	possibly additional; apply transitivity	possibly additional; UCCs in R remain iff each tuple in R joins max. one tuple in S (and vice-versa)
R.A = S.A (FK R $\rightarrow$ S)	dist(R.A)	prev. val. in columns from R	min(R.A)	max(R.A)	prev. val. in columns from R	mfv(R.A)	prev. val. in columns from R	— " —	no change in R
R.A = S.A (FK R $\rightarrow$ S) & ALL S.A included	— " —	prev. val.	— " —	— " —	prev. val.	— " —	— " —	— " —	no change in R; no new in S
UNION	dist(R.A) + dist(S.A) as boundary	n/a	MIN(min(R.A), min(S.A))	MAX(max(R.A), max(S.A))	n/a	unknown	n/a	any that exist in only one relation break; others may break	previous may break; possibly additional

information, such as keys and other constraints, is also useful in general, but is already reflected in the base statistics.

Base statistics. Database systems already collect valuable statistics for query optimization. For typical exploratory `SELECT * FROM table` queries without selection predicates these base statistics are already valid for the query result and can be displayed directly. For more complex queries, these statistics cannot be used directly but serve as a basis to efficiently calculate the metadata of the final query result.

Calculation. In some cases metadata can be obtained only through additional calculations. While iterating over all rows, one can keep track of values (or value combinations) to determine certain quantities or dependencies. For multi-column metadata, such as FDs and UCCs, additional calculations cannot be avoided, in general. Specialized algorithms can be executed at different nodes during query execution.

SQL. Any constant in the SQL statement can be used to derive metadata about the query result. For example, the predicate `WHERE A=3` already tells us that the maximum, minimum, and most frequent values are all equal to 3 in the query result, unless it is empty.

3.2 Piggyback collection strategy

The different phases of piggyback metadata collection during query execution are illustrated in Figure 2. The steps in Phase 1 are initialization steps that are executed in the leaves of the execution tree, Phase 2 is applied to each inner-tree execution node, and Phase 3 includes actions performed in the root node.

When reading data from a table (Phase 1), one can read the base statistics for the columns that are part of the result. We assume, for now, that these statistics gathered by the DBMS are correct. If no base statistics are available, one can either calculate them on-the-fly or continue without any base statistics. In Phase 2, depending on the current node, previous values can be *forwarded*, because the operator does not change them, or *updated* if their new value is known. In the case where changes occur but the new values cannot be determined easily, the previous values can be used as boundaries. Details on how different nodes are handled in this phase are provided in Section 3.3 and Table 1. Finally, Phase 3 is responsible for ensuring completeness and correctness of collected data and finalizing any missing values directly on the query result.

3.3 Metadata modification during execution

Two observations simplify metadata calculation: First, only changes in projected attributes, i.e., those that are part of the `SELECT` clause, need to be considered. Second, with exception of union and aggregation, we assume that query processing does not create new values in a given column; they are only removed or duplicated. The remainder of this section describes two examples of metadata modification, namely join and selection, followed by an approach on how

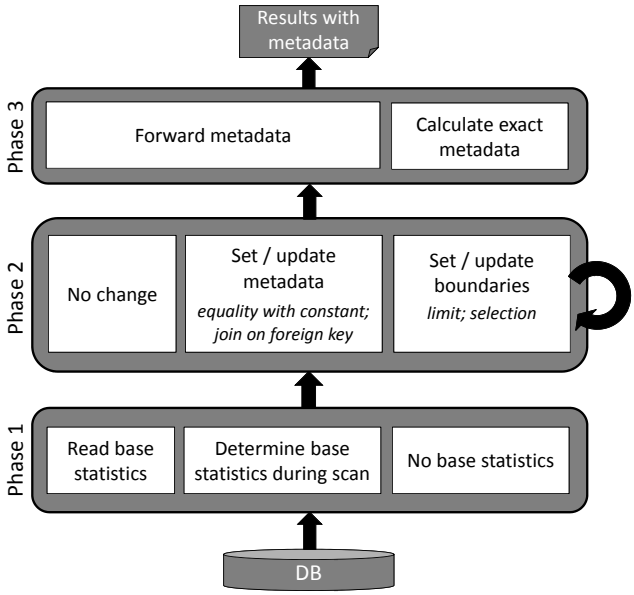

Fig. 2. Phases of piggyback metadata collection

these modifications can be incorporated in the piggybacking idea. A complete overview of how the most common operators modify result metadata is supplied in Table 1.

Join. Consider, for example, how the maximum value in a column `A` changes through a join on `R.A = S.A`. In general, tuples from both tables may be removed, because they do not have a join partner. The maximum in the joined column `A` is therefore the largest value that appears in both `R.A` and `S.A`, which is bound by the smaller of the two previous maximums. Thus, Table 1 shows "`MIN(max(R.A), max(S.A))` as boundary". If the join attribute `R.A` is a foreign key on `S.A`, all values of `R.A` remain in the result; the maximum does not change.

Selection. In some cases the metadata cannot be determined directly. For instance, a selection predicate `A >= y` (`y` being a constant value) cannot increase the number of distinct values in any column but it can decrease. The previous number of distinct values can be used as an upper bound. For column `A`, some additional information is available through the selection predicate: There can be only as many distinct values as exist in the range between y and the current maximum of the column (i.e., simply the difference for integers, but infinity for Strings and decimals). Therefore, the previous value or the range can be used as a boundary, whichever is smaller.

To make use of such boundary information we maintain validity information for every attribute-based metadata. That is, every node marks values that it may have changed as a boundary, or updates the metadata if possible. At the root node, during the final iteration over all tuples, the values of all columns

where one or more metadata are marked as boundaries are examined. As soon as the boundary value is found, the corresponding metadatum can be set to final. It may, however, happen that the boundary value is no longer present. In that case, when the result has finally been completely iterated, the true value has been determined by aggregating the observed values into separate data structures. If all metadata on the column are final, the values need not be checked anymore.

3.4 Dependencies in query results

As mentioned in Section 3.1, additional effort is needed eventually to determine multi-column metadata. For functional dependencies (FDs), for example, we use an approach similar to FDep [2]. Starting with all possible FDs as candidates, a hashmap of left-hand side and right-hand side values is maintained for each candidate, to check whether the current row contradicts it. This is the case if the current left-hand side is already paired to a different right-hand side. If so, the candidate is removed and no longer has to be considered. Additionally, some pruning rules are applied: If the distinct count of a column is known, we know it cannot functionally determine any column with a higher distinct count and cannot functionally depend on a column with lower distinct count. For the latter case, it is already sufficient to know that the upper bound of a column is smaller than the known value.

Other approaches, such as TANE [3], are known for their superior efficiency, but rely on a column-based approach, which is not well applicable for streaming (result-)data. Unique column combinations can be calculated similarly by taking all column combinations as candidates and removing them when a duplicate is found.

4 Implementation in PostgreSQL

To evaluate our conceptual ideas we extended the PostgreSQL DBMS for a subset of SQL. Our current implementation supports SPJ queries and determines *minimum*, *maximum* and number of *distinct values* for all individual columns of a query result, and determines all unary *functional dependencies* that hold for the query result. As described in Section 3.2, the metadata can be determined from the base statistics and/or during query execution. We prototypically implemented a subset of the rules from the large set in Table 1, namely the sort-, limit-, and selction-rules for min, max, distinct, and FDs.

Our implementation affects only the executor component of PostgreSQL's query processing pipeline. The executor processes the optimized query tree, made up of operator-nodes, and itself consists of an initialization phase, an execution phase, and a finishing phase. Our approach creates a global data structure to store metadata for each query and extends the first two phases.

1. Initialization phase The initialization phase prepares the optimized query tree for execution. During this phase we collect information that we can glean from

the query itself, and for now ignore the actual data. For instance, we collect information from selection predicates with numeric constants, as mentioned in Table 1. In the case of an equality predicate, the minimum and maximum value are the same for all result tuples, as long as there is at least one. There are other node types, such as sorting or hashing, that do not change any metadata.

If a node theoretically could change some of the metadata of the final result set, all of its affected columns and their corresponding metadata can be used only as boundaries as described in Section 3.2. In order to detect these *boundary* values in the execution phase, we flag all metadata that could change during query execution. PostgreSQL supports 31 different node types, including twelve scan-types and three join-types. Our implementation supports 20 of these node types; the others were not needed for any of our benchmark queries.

At the end of the initialization phase we fetch metadata from the base statistics. In PostgreSQL the base statistics are determined with fixed-size samples of the data. To obtain accurate results for all example queries, we increased the sample size from 30k (the default) to 30m. As we rely on the potentially inaccurate base statistics of PostgreSQL, the resulting metadata of our implementation can be inaccurate, too. However, in our experiments we did not experience any differences in the resulting metadata. To achieve perfect accuracy in the presence of inaccurate base statistics, one would have to determine the base statistics separately. Other DBMS calculate the base statistics differently. For instance, DB2 can be configured to calculate them accurately as well as on samples.

2. Execution phase In the execution phase each node of the optimized query plan processes the relevant tuples of the database. In this phase we update every metadata that was marked as "boundary" in the initialization phase. In each node PostgreSQL iterates over the tuples. In particular, we use this iteration in the root node for our updating process. This update is done by comparing every tuple of the result set with the boundaries. As soon as a boundary is reached, we cancel further comparisons for this particular attribute. For instance, if we know that a column has at most 300 distinct values, we cancel the counting of the distinct values as soon as we find the 300th one. We apply the same technique to the minimum and maximum value of a column. Additionally, we now determine all unary functional dependencies of the result set by using the incremental FD algorithm described in Section 3.4.

5 Prototype Evaluation

This section presents the evaluation results of our prototype as described in Section 4. First, we elaborate on the general benchmarking setup, then show the measured results, in particular query processing overhead, and briefly discuss them.

5.1 Experimental Setup

Our tests ran on a machine with an Intel Core 2 Duo E8400, 4GB DDR2 RAM, and a 160GB HDD. The operating system was Ubuntu 13.10 64bit and we used PostgreSQL build 9.3.1.

We compare regular PosgreSQL with our own approach (extended PostgreSQL), with and without the calculation of functional dependencies, as well as with a baseline approach. As baseline we run the query itself followed by a second SQL query with the original query as a subquery, calculating the same metadata as our implementation. The runtime of the two queries is added. The baseline queries are created with the following template:

```
SELECT min(sub.col1), max(sub.col1),
       count(distinct sub.col1),
       min(sub.col2), max(sub.col2),
       count(distinct sub.col2), ...
FROM   [subquery] as sub;
```

We ran those four approaches on two databases of different size, data origin (generated vs. real), and domain: TPC-H[4] (Scale factor 1 , approx. 1GB) and the MusicBrainz database (approx. 5GB) [8].

Of the 22 standard TPC-H queries we used the 16 queries that are supported by our implementation. We use an adjusted version of the benchmark for PostgreSQL[5], which rewrites aggregate subqueries with large tables to joins. Finally, we removed the `LIMIT` statements to adapt them more to our use case and make the queries more cumbersome.

MusicBrainz is an open-source database consisting of community-maintained music information. It fits our use-case of a database engineer exploring an unknown data set. For evaluation purposes we use eight different ad hoc, explorative queries (including scans, joins, selections, and aggregation) that are applicable for such a scenario. The list of these benchmark queries can be found in the appendix.

Every query of our workload for both TPC-H and MusicBrainz was executed five times, we report the average runtime. We measure the runtime for each query with the Ubuntu `time` command. To simulate the exploration on freshly acquired datasets we want to avoid caching effects and restart PostgreSQL and delete the system cache after every query, except for the baseline approach where we restarted after the second query.

Main memory consumption is not measured, but we acknowledge that it is high due to the additional data structures used in our implementation. We did perform the TPC-H benchmark with scale factor 5 (approx. 5 GB) with the piggyback approach including functional dependency calculation and it ran successfully on our testing machine.

[4] http://www.tpc.org/tpch/default.asp
[5] http://www.fuzzy.cz/download/tpch-queries.tgz

5.2 Benchmark Evaluation

The TPC-H benchmark results are shown in Figure 3. As a first observation, the baseline approach is always the slowest, but surprisingly with less than 50% overhead in most cases. One would expect more overhead by running two queries, but PostgreSQL benefits from caching of the first query. This caching apparently fails for Query 1, which includes many aggregations.

Our piggyback approach shows good results regarding the percentage overhead compared to regular PostgreSQL and the baseline. We benefit from the small result set sizes for most of the queries. The average execution time for the piggyback approaches was 35s and 37s, respectively. For TPC-H we achieved a maximum overhead of 3% without and 15% with the calculation of functional dependencies. In average our overhead is less than 1% without and 7,5% with functional dependencies calculation. With the 5GB TPC-H, the average overhead is reduced to less than 1% even with functional dependency calculation. This is due to the fact that most queries do not have a larger result set, so in relation to the much longer query execution, the overhead decreases.

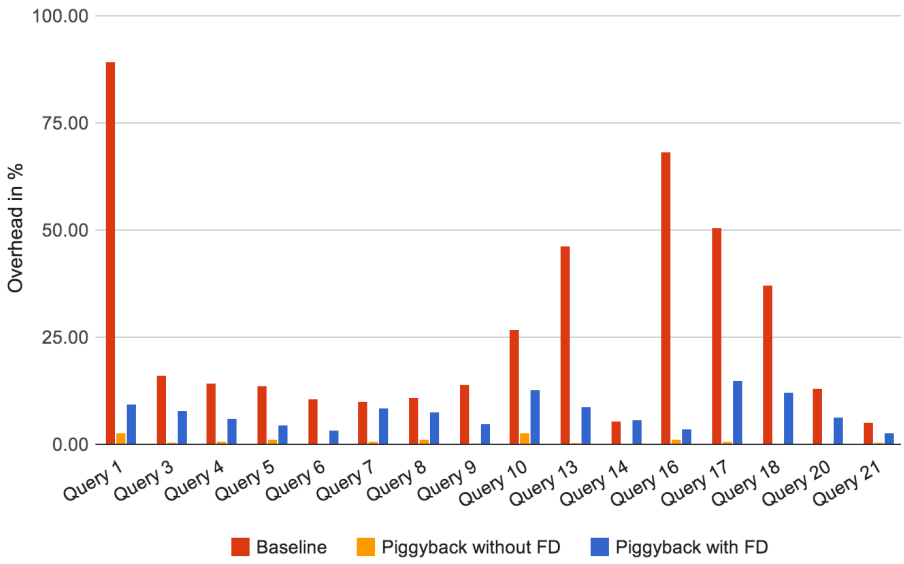

Fig. 3. TPC-H benchmarking results, showing the overhead compared to regular PostgreSQL

The performance chart for MusicBrainz (Fig. 4) shows the overall runtime of every query for each of the four approaches. The larger size of the MusicBrainz database with larger result sets leads to slower runtime compared to TPC-H. Again, the baseline approach is the slowest. Especially Query 7 with an aggregation on a large result set of more than 6 million rows has a slow runtime, similar

to Query 1 for TPC-H. Regarding Query 2 the runtime for our approaches (and the baseline) is very slow, because a join of the artists and recordings tables creates a result set of more than 12 million rows. Calculating metadata for all of its columns is very expensive, and in this case no base statistics can be used to improve the performance. In contrast, Query 1, which is a projection of all attributes from a smaller table, has a smaller result set and additionally can make use of base statistics. Thus, its runtime is much faster.

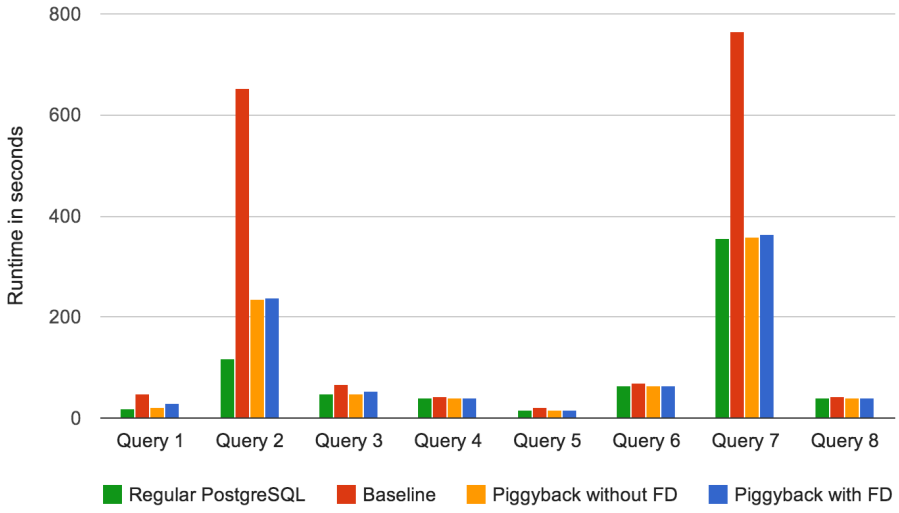

Fig. 4. MusicBrainz benchmarking results, showing the runtime of the four approaches

Apart from Query 2, the maximum overhead for our approach is 15% without and 60% with the calculation of functional dependencies compared to regular PostgreSQL. On average over all queries we achieved an overhead of 13.8% without and 20.8% with functional dependencies calculation.

We observe that the runtime depends on the result set size, data characteristics, and the nature of the query. Smaller result sets lead to a small overhead. Larger result sets, like that of MusicBrainz' Query 2, increase the overhead, but compared to the baseline it is still significantly lower. Also, even with large result sets, like MusicBrainz' Query 7 (all tracks, grouped by name), we can achieve fast runtime when base statistics can be used, or boundaries are found early.

Determining the functional dependencies for the result table has some additional overhead, which depends on the number of columns, and the existence of base statistics that can be used for pruning. Overall the the runtime of our approach is very good: across both benchmarks we achieved an average overhead of only 13% compared to regular PostgreSQL. In many use-cases this small overhead is well worth the effort, for instance when our piggyback techniques can

avoid follow-up exploratory queries altogether, or when it uncovers surprising characteristics of the data.

6 Conclusion and Future Work

We presented theoretical and practical concepts for enhancing query results with single- as well as multi-column metadata. The metadata can be collected from base statistics and are modified during query execution. These ideas are implemented in PostgreSQL; the average overhead of our approach is 13%. The code is available as a branch of PostgreSQL at `https://github.com/mpws2013n1/postgres/tree/piggyback_master`.

Our implementation can be extended for more metadata, such as most frequent values, unique column combinations, non-unary functional dependencies, etc. Another idea for future work is to use the collected metadata internally, e.g., for the query optimizer or following queries in the spirit of [9]. Finally, our approach to communicate the metadata to the client is for now ad-hoc. In general, an extension of the database driver protocols with additional metadata per column should be considered.

For our main use case, which is the exploration of an unknown database, the current overhead is acceptable. Nevertheless, the performance can be further improved, for instance by handling more special cases, calculating metadata earlier, and by allowing approximate metadata.

References

1. M. J. Carey and D. Kossmann. On saying "Enough already!" in SQL. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 219–230, 1997.
2. P. A. Flach and I. Savnik. Database dependency discovery: A machine learning approach. *AI Communications*, 12(3):139–160, 1999.
3. Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42(2):100–111, 1999.
4. D. A. Keim. Information visualization and visual data mining. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 8(1):1–8, 2002.
5. N. Khoussainova, M. Balazinska, W. Gatterbauer, Y. Kwon, and D. Suciu. A case for a collaborative query management system. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2009.
6. M. V. Mannino, P. Chu, and T. Sager. Statistical profile estimation in database systems. *ACM Computing Surveys*, 20(3):191–221, 1988.
7. K. Morton, M. Balazinska, D. Grossman, and J. Mackinlay. Support the data enthusiast: Challenges for next-generation data-analysis systems. *Proceedings of the VLDB Endowment*, 7(6):453–456, 2014.
8. MusicBrainz Database – MusicBrainz. `http://musicbrainz.org/doc/MusicBrainz_Database`, Feb. 2014.
9. M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO-DB2's learning optimizer. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 19–28, 2001.

A Musicbrainz Benchmark Queries

The following SQL queries were used for the performance evaluation on the Musicbrainz database (Section 5.2, Figure 4).

1.
```
SELECT  *
FROM    release;
```

2.
```
SELECT r.name, r.length, ac.name, ac.artist_count
FROM   artist_credit AS ac, recording AS r
WHERE  ac.id = r.artist_credit;
```

3.
```
SELECT *
FROM   track
WHERE  position = 10;
```

4.
```
SELECT name, position, length
FROM   track
WHERE  position = 5
AND    length < 30000;
```

5.
```
SELECT *
FROM   medium,artist_credit,release
WHERE  medium.track_count < artist_credit.artist_count
AND    release.artist_credit = artist_credit.id
AND    release.id = medium.release;
```

6.
```
SELECT (track.length-recording.length) AS diff
FROM   track, recording
WHERE  track.id = recording.id
AND    track.id < 10000;
```

7.
```
SELECT   name, max(position) AS max_position
FROM     track
GROUP BY name;
```

8.
```
SELECT  name, position, number
FROM    track
WHERE   length < position;
```