Design Of An Interface To Gather Execution Statistics Of A Dataflow Program

Seminar Paper by Matthias Sax

10th June, 2010

Mentor: Dipl.-Inf. Rico Bergmann
Contents

1 Introduction 1

2 Basics 3
  2.1 Adaptive Query Processing 3
  2.2 Pros and cons of different AQP approaches 6
  2.3 AQP in plan-based systems 7

3 Dynamic statistic collecting 11
  3.1 Statistic collecting techniques 11
  3.2 Pipelined vs. non-pipelined statistic collecting 13
  3.3 Making statistics available – Pull vs. Push 15
  3.4 Histograms 16
  3.5 Other statistics 21

4 Designing the statistic tracking interface 24
  4.1 Nephele 24
  4.2 Where to collect statistics 25
  4.3 Interface prototyping within Nephele 28
  4.4 Performance evaluation 29

5 Conclusion 32

A Performance evaluation table 33
# List of Figures

2.1 Design of an eddy operator. .................................................. 4  
2.2 Two dynamic query evaluation plans. ....................................... 9  

3.1 Example of a histogram. .......................................................... 17  
3.2 Equi-width and equi-height histogram for an integer domain from 1 to 50. 19  
3.3 Equi-width and cumulative histogram. ...................................... 20  
3.4 Example of a 2-dimensional histogram. .................................... 21  
3.5 Example of two operators exchanging data via a buffer. ................. 22  

4.1 Example of a JobGraph. ........................................................... 26  
4.2 Example of a possible (partial) ExecutionGraph for the JobGraph of 4.1. 27  
4.3 Average job execution times and difference in percentage. ............. 29  
4.4 Average job execution times and difference in percentage. ............. 30  

A.1 Job execution times in milliseconds for each run and evaluation. ....... 33
List of Tables

2.1 Model, goals, assumptions and approaches of the three system families (according to [BB05]) .......................... 5
Chapter 1

Introduction

Effective query processing has become more and more difficult due to larger amounts of data and more complex queries running on this data. A lot of research was done over the last years to overcome this problem. One concept is Adaptive Query Processing (AQP) [DIR07, GPFS02, BB05]. It is a promising approach for getting better performance in query processing if statistical information is not up to date or not available at query optimization time. One idea is to collect statistics during query execution. The collected values get compared with the estimated values from the optimizer in order to encounter plan sub-optimalities. The query execution plan gets corrected if a better plan is available.

Another way to handle big data sets efficiently is parallel data processing. Cloud Computing is a new approach for parallel data processing. The idea is to provide a homogeneous environment (the Cloud) to the user, hiding the details of a dynamic computer cluster. Dynamic means that the number and kind of the machines in the Cloud is neither fixed nor uniform. New machines can be seamlessly added to or dropped from the Cloud (e.g. if they fail). The system adapts to this changes automatically. The user just submits his task(s) to the system not knowing the hardware or the configuration of it.

The Map/Reduce paradigm [DG08] was developed to do parallel data processing in a very efficient way. Map/Reduce systems are processing a job, which consists of Map and Reduce tasks, by running many instances of each task in parallel. The task instances might be distributed over multiple machines in order to handle very big data sets in an effective manner. At the same time the data is partitioned over these machines, so that each instance of a task just deals with a very small portion of the data.

Nephele [WK09] is a Map/Reduce like system developed at the Technische Universität Berlin. In opposite to pure Map/Reduce frameworks, it is not limited to the Map/Reduce programming model. There is just one kind of task, which is not dedicated to be a Map or Reduce task. Additionally, Nephele is especially designed for Cloud Computing and can adapt to the dynamic nature of a Cloud.

In this work, we define a new interface for Nephele to gather statistical information during job execution. This statistics can be used for job re-optimization in an AQP manner later on. The questions to answer are what kind of statistics (e.g. cardinalities,
histograms etc.) can be collected and how to provide them to consuming applications. It is not obvious, what kind of statistics will be needed for re-optimization. Therefore the interface must be as flexible as possible to ensure the ability to include new statistics later on. At the same time it must be easy to add new statistics to avoid a high maintenance overhead. As Nephele is running in a Cloud (in contrast to a static system like a computer cluster), new statistical information (e.g. what machines (which hardware) are available or what is the workload on a machine) might need to be collected as well.

This work is structured as follows: The second chapter introduces Adaptive Query Processing (AQP), including a discussion about different AQP approaches. A few AQP systems are introduced as well, to point out pros and cons of different techniques. The third chapter discusses statistic tracking at runtime for AQP and points out related difficulties for each AQP approach. The different types of statistics needed for different re-optimization techniques are considered as well. Chapter four starts with a quick introduction into Nephele, including a discussion about the possibilities how to design a statistic tracking interface within the system. A section about the actual design of the implemented interface and a discussion why this design was chosen follows. The chapter closes with an analysis of the tests executed with the interface prototype. The conclusion summarizes the paper and gives an outlook about open issues and how they might be solved.
Chapter 2

Basics

2.1 Adaptive Query Processing

Query optimization has been done for more than two decades including a lot of research on this area. Nevertheless there are still many problems to be solved. The most common optimization techniques use a cost model for optimization. The optimizer has to find the plan with the smallest cost according to this model. The cost estimations usually depend on cardinality estimations of intermediate result sizes. Very often optimizers use statistics on the data to estimate these cardinalities. The statistics used, usually include table sizes (tuple counts) and histograms to estimate predicate selectivities. The problem hereby is that estimation errors might build up critically if complex queries have to be optimized. This leads to suboptimal query execution plans (QEP) and therefore to a weak query execution performance. Another critical point leading to badly optimized QEPs is the optimizer’s assumptions about the data, like uniform distributions or attribute independency. These assumptions do not hold very often and can also lead to bad cardinality estimations.

Adaptive Query Processing tries to address these problems. Systems that monitor query execution are very common. The gathered information is either used to re-optimize the currently running query or to improve the optimization of all following queries. Babu and Bizarro [BB05] distinguish between three different AQP approaches: plan-based, continuous-query-based (CQ-based) and routing-based systems. Plan-based and CQ-based systems use a cost-based optimizer which generates an execution plan. Additionally, there is a feedback loop, which triggers re-optimization of the plan, if plan sub-optimalities are encountered. Plan-based systems are a straightforward extension to classical cost based optimizers. Many plan-based systems are tracking statistics during query execution. This runtime statistics are either used to encounter sub-optimalities or for query re-optimization. Some systems use the statistics even for both.

The major difference between a CQ-based system to a plan-based system is that a CQ-based system executes long running queries and therefore uses data streams as data sources [ZRH04]. A data stream is a potential infinite data source. This is a new challenge for query processing. I would like to give an example. To join two data streams the join
operator must actually buffer all input tuples of both streams. Otherwise there might be missing results. A dropped tuple from stream A could still join with a new arriving tuple from stream B. As infinite buffering is not achievable, query execution is performed in a sliding window over the data. The result of a query is always calculated according to the current data in this sliding window. The re-optimization focus is not on wrong or outdated statistics like in plan-based systems, but on changes in the data stream characteristics. The system has to decide on different conditions, when the execution plan has to be re-optimized. At the same time CQ-based systems have to adapt to other system conditions like data arrival rates etc. as well. Another main difference is that CQ-based systems support pipelined QEPs only. The calculation of a buffered intermediate result would never be finished due to the infinite character of the input data streams.

Routing-based systems are very different. They do not have an optimizer or an execution plan. Instead they route the tuples through operators, whereas each operator is returning a result (or none if a tuple does not satisfy a selection predicate for example). The system outputs a tuple if it got routed through all operators successfully. The eddy approach \[\text{[AH00]}\] represents such a routing system. An example of it is shown in figure 2.1. The problem is that there are many different possible orders, in which tuples can get routed through the operators. To figure out the best order, the system sends different tuples in different orders to the operators. The system can therefore monitor the selectivities of the different operators. Predicates with high selectivity (i.e. a small output) will be placed in front of the routing order. If the best route\(^2\) is found, most tuples are routed according to this best routing order. A few tuples still get routed through other routes for monitoring purpose. If the data characteristics are changing, the system is able to encounter a new best route. The system changes the used routing order to the new best order. Of course, other orders get still monitored.

Table 2.1 shows the three AQP approaches, their goals and how they try to achieve it.

---

\(^1\)In CQ-based systems, statistical information is very often not available at query optimization time.

\(^2\)The best order corresponds to the best available QEP for that query.
### Table 2.1: Model, goals, assumptions and approaches of the three system families (according to [BB05])

<table>
<thead>
<tr>
<th></th>
<th>Plan-based</th>
<th>CQ-based</th>
<th>Routing-based</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Statistics at optimization time</strong></td>
<td>Missing or outdated statistics</td>
<td>No statistics at all</td>
<td></td>
</tr>
<tr>
<td><strong>Changes in data characteristics</strong></td>
<td>Not expected</td>
<td>Expected</td>
<td></td>
</tr>
<tr>
<td><strong>Changes in system conditions</strong></td>
<td>Expected</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Objectives</strong></td>
<td>Detect and correct cases where optimizer picks an inefficient plan</td>
<td>Ensure that plan is efficient for current input data characteristics</td>
<td>Ensure that tuples follow efficient routes</td>
</tr>
</tbody>
</table>

(a) Problems to solve

<table>
<thead>
<tr>
<th></th>
<th>Plan-based</th>
<th>CQ-based</th>
<th>Routing-based</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Optimizer</strong></td>
<td>Conventional optimizer extended for re-optimization</td>
<td></td>
<td>No conventional optimizer</td>
</tr>
<tr>
<td><strong>Re-optimization</strong></td>
<td>Postponing optimization decision to runtime</td>
<td>Monitoring query execution. Re-optimization gets triggered if plan sub-optimalities are encountered</td>
<td>Implicitly by exploring different routes. Greedy optimization via selective tuple routing</td>
</tr>
</tbody>
</table>

(b) How to achieve the goals

<table>
<thead>
<tr>
<th></th>
<th>Plan-based</th>
<th>CQ-based</th>
<th>Routing-based</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>QEPs</strong></td>
<td>Pipelined and non-pipelined plans</td>
<td>Pipelined only</td>
<td>Routes through operators simulate pipelined plans</td>
</tr>
<tr>
<td><strong>Support Data Streams</strong></td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

(c) Other differences
2.2 Pros and cons of different AQP approaches

There are different AQP approaches each focusing on different problems. All of them have certain advantages and disadvantages.

Using an optimizer

Plan- and CQ-based systems are using an optimizer and an execution plan. Cost based optimization has been around for many years and most today optimizers are doing a very good job if the required statistics are available and up to date. These optimizers are an excellent choice if you have to optimize complex queries which are nested or have correlated sub-queries in it. The plan space considered during optimization is quite big\(^3\). The optimized QEPs are usually very efficient, even if the searched plan space is incomplete. Some optimizers for example consider left-depth join orders only. The problem with this optimization is that it relies on statistics and intermediate results sizes estimations which might be wrong, outdated or even missing. Mistakes in cardinality estimations are building up from the leafs of the QEP\(^4\) (base table access) to the root (final result). As queries are getting more complex, the QEPs are getting deeper, which leads to bigger estimation errors. Another point is the quality of the statistics the optimizer is using. Tracking statistics is usually a big overhead and is therefore done offline in many systems. This means that cost based optimizers are mostly working on outdated statistical data. Additionally, it is getting harder to know in advance what kind of statistics will be needed on what data. As a result, the right statistical information might be missing at all.

Routing-based system in opposite make their decision on knowledge obtained at query runtime. Recall the section above, that the best route is found by exploring different routing orders through the operators at runtime. Mistakes due to missing or outdated statistics leading to wrong cardinality estimations cannot occur. The disadvantage is that optimization is usually done via a greedy strategy. The plan space considered is therefore quite small, compared to the plan space searched by a cost-based optimizer. This may lead to sub-optimal routing strategies for complex queries.

Runtime overhead

There is a big runtime overhead for scheduling the tuples in routing-based systems. First, the information which tuple got routed to which operator must be logged. Second not every tuple can be routed to every operator. If the input is three tables \(A\), \(B\) and \(C\) and a join \(A \bowtie B \bowtie C\) should be performed. Tuples from table \(A\) cannot be routed to the \(B \bowtie C\) operator but just \(B\), \(C\) or \(AB\) tuples. Plan- and CQ-based systems do not have such a big runtime overhead. As they are using a QEP each tuple gets processed by each operator once and in the right order implicitly. Even if plan- or CQ-based systems are tracking some statistics, the runtime overhead is small. The disadvantage of plan-

\(^3\)Dynamic Programming and pruning strategies are often used, to search the (exponential growing) plan space in an efficient manner.

\(^4\)QEPs are often represented as trees.
and CQ-based systems are the context switches between (re-)optimization and execution which can happen multiple times. But if the executed QEP is optimal anyway, this disadvantage disappears. There is always a runtime overhead in routing-based systems, even if the data characteristics do not change and a single routing order is used (e.g. no ‘re-optimization’ has to be done).

2.3 AQP in plan-based systems

Many different adaptive processing techniques got developed over the last couple of years. This section introduces a few of these techniques. The two main classes of techniques are (1) systems, which gather statistical information during runtime, but use this information for following queries. LEO [SLMK01] is one example of such a system. (2) In the other type of systems, the feedback loop directly influences the execution of the currently running query. Many different approaches like Dynamic Query Evaluation Plans [GW89, CG94], Mid-Query Re-Optimization [KD98] or Progressive Optimization (POP, POP/FED) [MRS+04, KHM+06] have been suggested.

LEO

LEO [SLMK01] is a prototype for a LEarning Optimizer, implemented in DB2. The classical optimizer in DB2 is using statistics from the system catalog for optimization. These statistics get updated by the RUNSTATS command (usually started by the database administrator). As this updating process puts a heavy load on the system, it gets executed maybe once a day or even just once a week. This leads to outdated statistics and sub-optimal execution plans, chosen by the optimizer. The idea of LEO is to use the results of query execution, to ‘update’ the statistics in the system catalog. The original statistics of the system catalog are actually not modified. The monitored values get stored in a new catalog table. LEO is aware of this new table and uses the ‘old’ values together with the new feedback values to estimate the cardinalities for the query execution plan (QEP).

Every time a query gets executed, the QEP (including the estimated cardinalities) as well as the actual monitored cardinalities get stored in an analysis daemon. The daemon is aware of the former values it calculated and knows that the query optimizer used these adjustments already for query optimization. It takes this into account when calculating new adjustments. The daemon can either run online and provide a feedback immediately or offline as a batch job. The offline version has the advantage that it does not put a runtime overhead on the system. The disadvantage is of course, that the improved statistics are not available immediately. Immediately does not refer to the actual executed query as the feedback is always just provided to improve further query optimization. The

---

5. Results’ does not refer to the result of the query itself, but to e.g. cardinalities of intermediate results and other value which are useful for statistical adjustments.

6. This has the advantage to get a better re-adjustment, because the old adjustment values can be used to calculate the new ones.
runtime overhead for monitoring the query execution was proven to be minimal, as each operator just counts the number of tuples processed.

**Dynamic Query Evaluation Plans**

Dynamic Query Evaluation Plans [GW89, CG94] try to solve the problem of parameter markers in precompiled queries. Parameter markers make it very difficult (or even impossible) to make a good selectivity estimation for a predicate at query compile and optimization time. This may lead to far from optimal QEPs. But even if the query has no parameter marker, precompiling faces the problem of changing system conditions. An index which is used in a precompiled QEP could be dropped for example. Other possible system changes are heavily performed update/insert/delete operations on the queried data. A precompiled QEP cannot adapt to this changes either. The assumed data distributions and selectivities (etc.) at query optimization time are not valid any longer, and the QEP might be sub-optimal for the new conditions.

Dynamic Query Evaluation Plans are solving this problem by adding so called choose-plan operators into the QEP. The choose-plan operator is not a relational operator. It just links together alternative operators (and/or alternative sub-plans for sub-queries). Figure 2.2 shows two dynamic query evaluation plans. The first one is for the simple query:

```
SELECT * FROM table WHERE predicate = ?
```

The second one for the more complex query with a join:

```
SELECT * FROM t1,t2 WHERE t1.p < ? AND t1.key = t2.key
```

At query launching time the choose-plan operators get evaluated. Each operator decides on the current parameter marker values and system conditions what operator/sub-plan fits best. The resulting (optimal) QEP gets executed afterwards.

**Mid-Query Re-Optimization**

Mid-Query Re-Optimization [KD98] is a more dynamic technique than Dynamic Query Evaluation Plans. Dynamic Query Evaluation Plans postpone some optimization decision to query launching time. But after starting query execution, no more feedback is given to the system. Instead Mid-Query Re-Optimization is monitoring query execution directly. If plan sub-optimalities are encountered, query execution stops and re-optimization is triggered.

Monitoring of the query execution is done by the statistic collector operator. This operator is part of the QEP and gets executed like any other operator. The statistic collector operator is a pipelined operator which does not modify any data. It can collect any kind of statistic\(^7\) on the data flowing through it.

During query execution, the gathered statistical values are compared to the optimizer’s estimations. If sub-optimalities are encountered (i.e. the tracked value is off the estimated one by a specific fraction), re-optimization might be triggered. Whether it gets triggered or not, depends on certain factors like estimated time to finish the current plan, estimated

---

\(^7\)There is one limitation: As the operator is a pipelined operator, it just can collect statistics calculated by a single pass over the data. This in not severe in practice, as the most common statistics used are satisfying this pre-requirement.
Progressive Optimization

Progressive Optimization (POP) [MRS+04] is similar to Mid-Query Re-Optimization. It also uses a new operator (CHECK) which is part of the QEP and which can trigger re-optimization during query execution if plan sub-optimalities are encountered. The difference to Mid-Query Re-Optimization is the way, re-optimization gets triggered. Each CHECK operator has an associated value range. This value range is calculated using the optimizers cost model. If the tracked value is not within this calculated range, it is sure that the plan is sub-optimal according to the optimizers cost model. Hence the QEP is known to be sub-optimal and re-optimization gets triggered.

There are also differences in the re-optimization step. If re-optimization gets triggered, the current plan execution terminates and the operators are not allowed finish their work. Additionally, the optimizer does not get forced to reuse the intermediate results already calculated, but has the choice to use them or not. The optimizer decides this, based on its regular cost model for optimization\textsuperscript{8}. The optimizer is re-optimizing the whole query

\textsuperscript{8}Indeed there are examples for getting a sub-optimal QEP by reusing intermediate results.
and not just the remainder which got not executed yet. If no intermediate result can be used, query execution will start from scratch.
Chapter 3

Dynamic statistic collecting

3.1 Statistic collecting techniques

Dynamic statistic collecting can be done in multiple ways. Babu and Bizarro [BB05] list four ways of statistic tracking: observation, exploration, competition and profiling. This section will discuss the advantages and disadvantages of each technique.

Observation

Observation is a monitoring technique and often used in plan-based systems. In section 2.3 we already discussed three systems which are using observation. LEO [SLMK01], a very simple monitoring approach, is counting cardinalities of intermediate results. Mid-Query Re-Optimization [KD98] and Progressive Optimization [MRS+04] are introducing a new (non-relational) operator which gets included into the QEP. The operator is executed as any other operator in the QEP and can collect any kind of statistic. But observation cannot collect statistics on all data. It is limited to the QEP’s structure. E.g. it is not possible to track the join fan-outs of an alternative join order. Imagine the star-join-query `SELECT * FROM S,A,B,C WHERE S.a = A.a AND S.b = B.b AND S.c = C.c`.

Let the chosen join order be `(((S $\bowtie$ A) $\bowtie$ B) $\bowtie$ C)`. An alternative join order would be `(((S $\bowtie$ C) $\bowtie$ B) $\bowtie$ A)`, but we might have no chance to track the fan-out of `S $\bowtie$ C` in the original plan. This may happen, if re-optimization gets triggered before the last join starts execution. This may happen if the estimated cardinality of `S $\bowtie$ A` or `S $\bowtie$ A $\bowtie$ B` was wrong. This limitation has two consequences: (1) It limits re-optimization. If the optimizer estimates the join-fan out of `S $\bowtie$ C` wrong, it has no chance to encounter this error. Even if re-optimization gets triggered, another sub-optimal plan might be produced. (2) The statistics most often gathered are correlated, e.g. the selectivity of the second select-operator in a chain of two select-operators. Not all input tuples will be processed by the second operator but just the tuples satisfying the first operator’s predicate. Therefore just the selectivity of a subset of all input tuples is tracked, which is correlated to the first predicate. Gathering uncorrelated statistics is just possible in the leafs of the QEP.

The advantage of observation is that the statistical information is highly accurate.
Those statistics are used to optimize one single query\(^1\). Histograms designed to optimize one single query will always outperform those that are build to suite multiple queries. This leads to a higher quality of the statistical data compared to a histogram which will be used to optimize many different queries. This makes re-optimization more accurate, as the optimizer is usually using the statistics collected at runtime. The accuracy of the collected statistics also enables systems (see the CHECK operator [MRS+04]) to trigger re-optimization just if it is sure, that the current plan is sub-optimal. That avoids unnecessary re-optimization steps.

**Exploration**

Exploration is a technique used in routing-based systems. Tuples get routed over different routes to calculate e.g. different selectivities. The advantage in opposite to observation is that statistics can be collected on any data. Recall the example above: observation might not be able to estimate the join fan-out of \( S \bowtie C \) if the current join order is \(((S \bowtie A) \bowtie B) \bowtie C\). But exploration on the other hand can. It would route a fraction of \( S \) and \( C \) tuples to the \( S \bowtie C \) operator. If the fan-out is smaller than the \( S \bowtie A \) fan-out, it will switch the routing order to start with \( S \bowtie C \) instead of \( S \bowtie A \). Additionally exploration is theoretically never limited to gather correlated statistics as observation is. Every operator might be the first operator a tuple gets routed through.

The disadvantage of exploration is the exponential growing number of possible routes caused by an increasing number of query operators. In practice this growth makes it impossible to explore every route. The quality of the statistics is therefore highly dependent on the routes chosen for statistic tracking. In practice the big number of possible routing orders makes it also impossible to gather only uncorrelated statistics on the data. Another drawback is the fact that just a fraction of tuples gets sent along different routes. If this fraction is too small, the accuracy of the statistics gathered is rather low. If the fraction is too big, query execution gets slowed down unnecessarily. Exploration is therefore a sampling technique and the quality of the statistics is always depending on the fraction of tuples which is used.

**Competition**

Competition is similar to exploration. In opposite to exploration, where different tuples get routed over those routes, the same set of tuples gets routed over different routes. The operator selectivities for example, are therefore comparable without statistical errors. As competition is more a monitoring than a sampling technique the accuracy of the statistics is very high.

Also competition faces the problem of an exponentially growing number of possible routing orders by a growing number of query operators. Another big disadvantage is the redundant processing of a high number of tuples. It puts a big runtime overhead on query

---

\(^1\)This is actually a general advantage of dynamic statistic collecting if the aim is re-optimization of the current query. If the focus is on better optimization of following queries, this advantage only applies if one query gets optimized and executed multiple times.
execution. Additionally duplicate handling has to be done as well. As the same set of
data gets sent along multiple routes, duplicates could appear and must be deleted.

**Profiling**

The last approach discussed here is profiling. To collect statistical information about the
operators, a certain set of tuples gets routed through all operators independently. By
sending tuples through single operators, an exponential growing number of possible routes
(as in exploration and competition) is avoided. The complexity is linear in the number
of query operators. The accuracy of the statistical data is very high and only depends
on the fraction of tuples sampled. The gathered statistics are of course never correlated
statistics—it is not possible to get correlated statistical information with profiling. An-
other advantage of profiling is that it allows calculating any kind of statistics needed for
optimization.

The main disadvantage is the big overhead in statistic calculation. Profiling is not
part of query execution, but rather an independent step. The only way to minimize the
overhead is by using a smaller fraction of tuples. But this may lead to a low quality of the
statistical data.

### 3.2 Pipelined vs. non-pipelined statistic collecting

Operators in a QEP can be distinguished in pipelined and non-pipelined (blocking) opera-
tors. A pipelined operator starts its execution right after the first input tuple is available.
The pre-requirement for an operator to be pipelinedable is the independency of tuple pro-
cessing. The result of processing a single tuple is the same, whether the tuple is the first,
last or any other tuple being processed (e.g. evaluation of a predicate). Non-pipelined
operators in opposite, are buffering input tuples before they start processing the input. It
blocks processing until the input is completely available. An example for a non-pipelined
operator is the sort operator. The merge-sort algorithm for example, needs the complete
input before it can start working. The result of a blocking operator depends on all input
tuples. Pipelining usually performs better than non-pipelining. That is the main reason
why tracking statistics is usually done in a pipelined manner. The runtime overhead is
much smaller. However, this approach limits tracking statistics to statistics which can be
computed in a single pass over the data. Luckily in practice this is not a severe limitation,
as most statistics do not need more than one pass over the data.

Tracking statistics in a pipelined way faces another big problem. The collected statistic
might be available to late, i.e. the execution of the query is almost finished. Imagine two
operators A and B which are pipelined. If statistics are collected on the output of A in a
pipelined way, operator B cannot use this statistic until operator A has finished its work.
But at this time operator B is almost finished as well. Hence the statistic is not helpful
for B any longer. One solution could be, to break the pipeline, i.e. buffer the results
(the pipelined operator gets transformed in a non-pipelined operator). But this idea slows
down query execution dramatically. Of course this is just a problem for re-optimization
of the current query. If the statistics gathered are used to optimize following queries (like
in LEO [SLMK01]—see section 2.3), pipelined statistic tracking can be used without any restriction.

In [MRS+04] Markl et al. are introducing five techniques to tackle the problems of statistic tracking.

Lazy Checking

Lazy Checking\(^2\) (LC) means, that statistics are tracked at a materialization point. As discussed above, some operators are blocking operators. The (intermediate) result gets buffered (materialized) before execution gets started. If a statistic gets tracked at this point, it is available before the blocking operator starts its execution. At the same time it does not slow down query execution as the intermediate result gets materialized anyway.

Lazy Checking with Eager Materialization

Lazy Checking with Eager Materialization (LCEM) is a brute force solution. It breaks the pipeline and forces materialization. The reason why this technique might be necessary is the rare occurrence of native non-blocking operators. Whether it is necessary to use LCEM depends on the QEP. Despite the high runtime overhead, LCEM faces the problem of big intermediate result sets. It might not be possible to buffer those very big intermediate result sets. Even if it is possible to buffer there is still the risk of underestimating the cardinalities for intermediate results. If this happens, the buffer for materialization might be too small to receive all tuples. Therefore LCEM should be used very carefully.

Eager Checking without Compensation

Eager Checking is a pipelined tracking technique. The problem with this technique is that some tuples could already be returned to the user, when re-optimization gets triggered. To avoid duplicates in the output, an extra compensation step is needed. Eager Checking without Compensation (ECWC) means, that gathering statistics is a pipelined process, where a following operator must be a blocking operator. This blocking operator ensures that no result tuple gets returned to the user and no compensation is necessary. Of course, the gathered statistics are just useable for operators following this blocking operator.

Eager Checking with Buffering

Eager Checking with Buffering (ECB) is a technique to trigger re-optimization. The idea of ECB is that re-optimization can be triggered even before all tuples are processed. After a certain amount of tuples has been processed, it becomes clear whether the plan is sub-optimal or not. Those tuples gets buffered in the operator. After the threshold is reached and if no re-optimization is necessary, the buffered data are streamed to the next operator. From that point in time, any following tuples are not buffered any more. This leads just

\(^2\)Checking refers to the CHECK operator introduced in [MRS+04]. The idea of LC is however not limited to the CHECK operator, but can be applied to other statistic tracking operators as well. The name is just kept for convenience.
to a small latency. If, on the other hand, re-optimization gets executed, the buffering of the tuples ensures, that no result is returned to the user as query execution is blocked. A compensation step for handling duplicates is not necessary.

**Eager Checking with Deferred Compensation**

Eager Checking with Deferred Compensation (ECDC) can be used in so called SPJ-queries. These queries only contain selection, projection and join operators. To ensure high performance in query execution, blocking and buffering is strictly avoided. All result tuples returned to the user are buffered in a side table. If re-optimization occurs, the compensation step to avoid duplicates in the result set is done by an anti-join\(^3\) in the last step of the new QEP. This anti-join is done on the side table and returns the actual result of the new QEP.

### 3.3 Making statistics available – Pull vs. Push

After statistics are collected, they must be made available to other system components (e.g. to the optimizer), for further usage. The statistic tracker component is the producer of information (data) in this model, while the optimizer is one of the possible consumers. There are two approaches for making statistics available. (1) The producer is active and informs the consumer whenever new statistics are available. (2) Or the producer is passive and waits until the consumer requests new statistics. The advantages and disadvantages of both approaches (active producer/passive consumer vs. passive producer/active consumer) will be discussed in this section.

**Pull**

In the pull approach the producer, i.e. statistic tracker (ST), is the passive component. The ST provides an interface to the other system components. This interface might be a method to be called returning the statistic collected. The consumer, e.g. the optimizer, is the active component. By using the provided interface it pulls the statistic out of the ST. In section 2.3 we discussed LEO [SLMK01] which is using the pull approach. The gathered statistics are written into the system catalog. The optimizers can access the catalog tables and use the statistics to optimize queries.

The advantage of this approach is that multiple components can easily pull out the statistic information. There is no maintenance overhead for the producer if the consumer’s design gets changed, if new consumers are added or old ones get removed. It is also very easy to change the producer component as long as the pull-interface stays the same. The disadvantage is that pulling the statistics might cause some unnecessary overhead, e.g. if a component needs the complete (and not a partial) statistic, and so has to call the ST multiple times to check if new statistics are already available.

\(^3\)An anti-join of table \(A\) and \(B\) returns a \(A\)-tuple if this tuple does not have a join partner in \(B\).
Push

In the push approach the producer, i.e. statistic tracker (ST), is the active component. It knows what consumers exist and pushes the collected statistic to them (e.g. by sending a message). This does not mean that the statistic itself has to be included in the message (even though this would be possible). But a link to the file where the statistic is saved could be included in the message. In section 2.3 we discussed Dynamic Query Evaluation Plans [GW89] and the CHECK operator [MRS+04] which are using the push approach to trigger re-optimization. During statistic collection, the gathered values are compared to the optimizer’s estimations. If plan sub-optimalities are encountered, re-optimization gets triggered, i.e. the optimizer gets called. In the push model the consumer is always the passive component, waiting to be activated by the producer.

The advantage of this approach is that the potential overhead of calling the producer multiple times is avoided. The consumer is just waiting until the statistical data are available and it gets called by the producer. In the case, where re-optimization gets triggered there is one more advantage: re-optimization can get triggered immediately after plan sub-optimalities are encountered. In the pull model it would take some time until the statistics get pulled out again for comparison. This would delay the point where query execution stops and re-optimization starts.

The disadvantage of this approach is that comparing the current values with the estimated ones is done continuously, what results in an additional runtime overhead. Furthermore it is more difficult to add or remove consumers to such a system. The producer must be adjusted to those changes (which is not necessary in the pull approach).

3.4 Histograms

Histograms are the most common tool to estimate predicate selectivities and join fan-outs. Before histograms were used, one way of cardinality estimation was to save the max- and min-value of an attribute as well as the number of tuples in a table. Selectivities were estimated using the uniform data distribution assumption (i.e. \( \text{cardinality}_{\text{estimated}} = \frac{\text{max} - \text{min}}{\text{count}} \)). But these estimations are sub-optimal in many cases.

Histograms [Ioa03] are used to make much better estimations about selectivities and are built over one (or multiple) attribute(s). The domain of an attribute is the set of possible values the attribute can have. By building a histogram the domain of an attribute gets partitioned into non-overlapping intervals, called buckets\(^4\) (i.e. each bucket has an associated start- and end-value which are building the bucket’s range). A histogram is a mathematical object consisting of these buckets whereas each bucket holds the number of tuples, which attribute’s value (the histogram is build on) is within the bucket’s range.

For a given table \( T \) with an attribute \( \text{att} \) of domain integer, one bucket’s range could be \([0; 100]\). The stored value in the bucket is the number of tuples in \( T \), where the value of \( T.\text{att} \) is between 0 and 100 (100 exclusive). If a predicate \( T.\text{att} = 23 \) is specified, the result’s cardinality is estimated using the bucket’s value. This may happen conservative

\(^4\)The number of buckets is always finite as all common domains are finite as well.
CHAPTER 3. DYNAMIC STATISTIC COLLECTING

Estimated cardinality without using a histogram.

<table>
<thead>
<tr>
<th>Bucket</th>
<th>Bucket Count</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bucket 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bucket 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bucket 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bucket 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bucket 5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.1: Example of a histogram.

using the bucket’s value as an upper bound for the result’s cardinality. If the domain holds discrete values a more aggressive estimation can be made. One can use the *continuous value assumption*\(^5\) and assume a uniform data distribution within a bucket. The cardinality can be estimated like \(\text{cardinality}_{\text{estimated}} = \frac{\text{number\_distinct\_values\_in\_bucket}}{\text{bucket\_count}}\). This more aggressive estimation can also be made if the domain is not discrete. But the number of distinct values within the bucket must be known. It could be stored within a bucket together with the actual count value. An example of a histogram is shown in figure 3.1.

There are many different types of histograms and the most common types are introduced in this section.

**Equi-width**

Equi-width histograms are the simplest form of histograms where each bucket has the same range. For example a histogram for an attribute \(\text{att}\) of a \(\text{CHAR}(5)\) domain\(^7\) could have 26 buckets with ranges \([A; B], [B, C], \ldots [Z, ZZZZZ]\). The first bucket holds the number of tuples which \(\text{att}\)-values start with A, the second bucket hold the number of tuples which \(\text{att}\)-values start with B, and so on. If a predicate \(T.a = 'MATH'\) is specified, the result’s cardinality is be estimated with the \([M, N]\)-bucket’s value.

The trade off, equi-width histograms are facing, is to decide how many buckets should be used. As more buckets are used, as more accurate the estimations are. But at the same time the size of the histogram is growing linear in the number of buckets. As domains are usually quite big, the size of a histogram might grow critically if too many buckets are used.

The biggest problem of equi-width histograms is that there might be a big skew within a bucket. Imagine a bucket with a range of 100 distinct values. It could be that the bucket’s

---

\(^5\)The *continuous value assumption* mean, that all values of the bucket’s range are used.

\(^6\)The value \(\text{number\_distinct\_values\_in\_bucket}\) is not necessarily the actual number of distinct values in the bucket. It could also be the number of possible distinct values according to the attribute’s domain: \(\text{number\_distinct\_values\_in\_bucket} = \text{bucket\_upper\_bound} - \text{bucket\_lower\_bound}\).

\(^7\)\(\text{CHAR}(5)\) are assumed to hold strings with 1 to 5 characters which are just capital letters from A to Z.
count results from just one single value which is in the bucket’s range. This would lead

to a bad estimation for 99% of the bucket’s values. Additionally, empty buckets would be

possible. Empty buckets would not lead to bad estimations, but they could result in a big

memory overhead.

Another problem is that the data distribution is usually not uniform for the whole
domain. This results in empty buckets on the one hand and buckets with a very high
count on the other hand. These diverse counts in the buckets, may lead to a big skew
within the histogram. The estimations are getting worse as higher the counts in the
buckets are. Imagine a histogram over dictionary entries. It might be a good idea to have
many buckets for words starting with S (e. g. ‘Sa’ to ‘Sz’), but just one bucket for all
words starting with J and K. Equi-height histogram (see next paragraph) try to solve this
problems.

**Equi-height**

Equi-height (or equi-depth) histograms do not have uniform bucket ranges. The ranges

are chosen such that the number of tuples belonging to each range is roughly equal. Recall

the example above with the dictionary entries. In this case there might be just one bucket

for all words starting with J or K ([J, L]) but multiple buckets for words starting with S

(i.e. [S, SB[, [SB, SC[, ... [SZ, T]). Hence skews within a bucket are smaller than for

equi-width histogram leading to better cardinality estimations. Equi-height histograms

also avoid buckets which are empty, reducing the histogram’s size.

Figure 3.2 shows an equi-width and equi-height histogram on the same data. The

equi-height histogram improves the upper bound estimation for 30 values [21 to 50] (the

upper bound goes down) and makes it worse for 10 values [11 to 20] (the upper bound
goes up). The aggressive estimation gets improved for 25 values but makes it worse for 15
values. Nothing changes for the first 10 values for both estimations.

Even if the estimations are better using equi-height histograms, it is more difficult to
build them. The problem is that the optimal bucket ranges are not known initially. One

possible approach is to start with the ranges of an equi-width histogram. If the count of a

bucket reaches a certain threshold the bucket gets split into two. The range gets divided

in the middle and the two new buckets get half of the current count assigned. But this

splitting technique may result in big skew as well. Imagine a bucket with integer range

[0; 100]. Let all values within the buckets be 23, and the count threshold 10,000^8. After

splitting we would have two buckets with ranges of [0; 50] and [50, 100]. Both buckets have

a count of 5,000, whereas the right counts are 10,000 and zero respectively.

A better way to build an equi-height histogram is to sort the data on the histogram’s
attribute beforehand. The lower bound for the first bucket would be the first value of the
sorted data. As the height of the histogram is known, height tuples get read. The value of

the heightth tuple is the upper bound for the first bucket. This process can be repeated

until the end of the data is reached.

The problem of this building process is, that sorting the data slows down building the
histogram. The complexity for building the histogram changes from O(n) to O(n log n)

^8This would imply a target hight of 7,500 per bucket.
CHAPTER 3. DYNAMIC STATISTIC COLLECTING

(a) Equi-width histogram with range size 10.  
(b) Equi-height histogram with bucket size 10.

Figure 3.2: Equi-width and equi-height histogram for an integer domain from 1 to 50.

where \( n \) is the number of tuples. This implies that a single pass on the data is not sufficient any longer to build the histogram. This can be critical from an AQP point of view, because tracking statistics is preferred to be done in a pipelined way.

Cumulative Histograms

Cumulative histograms got developed to estimate cardinalities for range queries. Range queries do not have predicates which are specified for a specific value like \( \text{att}=100 \). Instead they have an upper or lower bounds for values (e.g. \( \text{att}<100 \)). Even if equi-width and equi-height histograms are useable to estimate range-queries, there is calculation overhead. The sum of multiple buckets needs to be built. Cumulative histograms are pre-computing these sums and store them as bucket values. A quick example will illustrate on an equi-width histogram how it works. Let the equi-width histogram have 5 buckets with ranges
A big advantage of cumulative histograms is that it is easy to re-calculate the original values of the equi-width histogram. It is just the difference of the current bucket and its predecessor (left neighbor). In our example, the fourth bucket with range $[15, 20]$ has count 6. This count is exactly the difference of the fourth bucket (18) and the third bucket (12) of the cumulative histogram. This invariant makes cumulative histograms very useful. They avoid the calculation overhead for range queries by being still usable for regular predicates. This invariant also enable the histograms to estimate range queries with upper and lower bound (e.g. $\text{att}>6$ and $\text{att}<19$). In the example above the estimated value would be $18 - 4 = 14$ what is the difference of the fourth and first bucket of the cumulative histogram. It is equal to the sum of the second, third and fourth bucket of the equi-width histogram: $14 = 6 + 2 + 6$.

The big disadvantage of cumulative histograms occurs by estimating disjunctive queries (i.e. queries with or-terms). In a regular equi-width histogram, it is easy to sum up the values of the relevant buckets. Using a cumulative histogram has the disadvantage that the bucket differences need to be calculated first for every single or-term.

**Multi-dimensional histograms**

The histograms discussed so far are histograms over a single attribute. In real systems conjunctive queries (i.e. queries with an and-operator) are very common. If a histogram for each involved predicate is available, both predicate selectivities can be estimated inde-
pendently. A global estimation is calculated (assuming attribute independency) by multiplying both selectivities. The drawback of this approach is that attribute independency does not hold very often what results in bad selectivity estimation. To overcome this problem, multi-dimensional histograms, i.e., histograms over multiple attributes, can be built. These histograms have multiple range axes (one for each attribute). A 2-dimensional histogram for example, has 2 range axis and span a 2-dimensional vector room. Each field in the vector is a histogram bucket in which the corresponding tuple count is stored. Figure 3.4 shows an example of a 2-dimensional histogram.

![2-dimensional histogram](image)

The problem is that the number of possible multi-dimensional histograms grows exponentially with the number of columns per table. In real systems, it is very difficult to predict the workload. That makes it practically impossible to build the right multi-dimensional histograms beforehand (offline). Hence the most commercial systems do not support multi-dimensional histograms.

In the AQP case however, we would like to build histograms on the fly, i.e., during query execution. In this case a specific query is given and building a multi-dimensional histogram is simple as the relevant attributes are known.

### 3.5 Other statistics

In the last section we discussed histograms. Building histograms is unfortunately expensive (and done offline in many systems). At the same time, the gained pros of histograms are not necessarily essential for AQP. We discussed already some systems where simple cardinalities of intermediate results got measured (see 2.3). The cardinality of an (intermediate) result is the number of tuples (or records) in the result\(^9\). Therefore it can be

\(^9\)The term *cardinality* comes from theory of sets and relational algebra. In relational algebra, a result is defined as a set of tuples. In theory of sets the cardinality of a set is the number of ‘items’ in the set.
measured easily by counting the result tuples. The runtime overhead is expected to be minimal what is reconfirmed in [SLMK01].

Histograms are often used to estimate selectivities. But selectivities can be measured directly as well. The selectivity of an operation is defined as the fraction of actual output tuples to all possible output tuples. The selectivity is therefore always a number between 0 and 1. The selectivity $s$ of a selection operator $p$ is simple defined as the fraction of output tuples to all input tuples (as the number of all possible output tuples is equal to the number of input tuples): $s_p = \frac{|O|}{|I|}$ (whereas $I$ is the set of input tuples and $O$ is the set of output tuples). The selectivity of a join operator is more complex. It is defined as $s_J = \frac{|O|}{|T_1| \cdot |T_2|}$ (whereas $O$ is the set of output tuples and $T_1$, $T_2$ are the input tables which are joined). The base of the fraction is the size of the Cartesian Product of $T_1$ and $T_2$ ($|T_1| \cdot |T_2| = |T_1 \times T_2|$), as the Cartesian Product is the biggest possible result.

Both selectivities (selection- as well as join-selectivity) can be calculated easily by counting input and output tuples.

**System statistics**

Additionally to statistic over the date, system statistics can be gathered. This includes CPU utilization, memory consumption, buffer occupation as well as data arrival rates for distributed or streaming systems.

In distributed systems it is important to balance the workload in a way that each CPU has about the same utilization. A balanced workload maximizes the system throughput. Operating systems usually provide an interface where CPU utilization can be monitored.

Memory consumption is also an important issue. If e.g. an intermediate result has to be buffered it is important to know if it fits into main memory. Otherwise it might be necessary to buffer the result at disk what involves expensive I/O-operations. Overestimation of the result size on the other hand results in allocation of too much memory. The unused main memory cannot be use by other operators. They are forced to wait until enough main memory is available or to do expensive disk operations. Monitoring the fraction of used memory to allocated memory and evaluation these numbers may therefore improve further main memory allocations.

This is also true for buffer occupation. Buffers are used to avoid idle times for operators.
Imagine two operators $Op1$ and $Op2$ where operator two is consuming the output of operator one. If e.g. operator one produces a lot of results at a certain time and operator two is not able to consume all the results immediately, $Op1$ would have to wait until $Op2$ has caught up. If a buffer is in between the both operators however, $Op1$ can write the results into the buffer and does not have to wait for $Op2$. At a later point in time, $Op2$ might catch up or even consume more input than $Op1$ can produce. But as $Op2$ is reading from a buffer it does not have to wait until $Op1$ produces the next result but can simply consume $Op1$-results from the buffer. Figure 3.5 shows a simple example with a buffer for 10 tuples. $Op1$ and $Op2$ have just to wait if the buffer is full or empty respectively. Therefore it might be a good idea to have big buffers. But if a buffer is never filled completely, it is a waste of main memory. Monitoring buffer occupations (and resizing a buffer if necessary) may therefore improve the throughput of a system.

Another important point is data arrival rates in distributed and streaming systems. Transferring data over the network is slower in an order of magnitude than performing disk operations. Waiting for data transferred over the network slows down query execution dramatically. Monitoring data arrival rates can help to get a better system throughput. If e.g. one operator has to wait for data very often, it might be a good idea to open a second connection to increase the bandwidth. On the other hand an operator may perform badly and cannot consume all arriving data. Starting new operator threads to increase parallelism might speed up query execution in this case (if parallel query execution is possible, e.g. for a select operator).
Chapter 4

Designing the statistic tracking interface

As we have seen, there is a big variety of (re-)optimization concepts and statistic tracking techniques. The goal of this work is to define an interface, which is flexible enough to support any kind of statistic tracking. This makes the interface independent of the actual (re-)optimization concept which might be used later on. At the same time it must be easy to add new statistics, to avoid a big maintenance overhead. Nevertheless, the runtime overhead must be as little as possible, as we do not want to slow down query execution.

Before the designed interface is described, a quick introduction to Nephele is needed to make the context of the work explicit. This is necessary to understand the chosen design.

4.1 Nephele

This section gives a quick overview of Nephele [WK09]. Nephele is a Map/Reduce like system. In opposite to pure Map/Reduce frameworks, it is not limited to the Map/Reduce programming model. There is just one kind of task, which is not dedicated to be a Map or Reduce task. Nephele’s design is similar to the Dryad system [IBY*07], but is especially developed for Cloud Computing environments. The input is a dataflow program which is represented as a (connected) acyclic directed graph (DAG). The vertexes of this input DAG are user defined tasks. The processed data is flowing from one task to the next along the defined edges. Each task may have an arbitrary number of input and output edges. The input DAG, which is called JobGraph, is the logical representation of the dataflow program.

Nephele takes the JobGraph as input and transforms it into an ExecutionGraph. The ExecutionGraph is Nephele’s internal data structure for executing the job and is no longer a DAG. Nephele has certain possibilities for transforming a JobGraph into an ExecutionGraph. Each task can get instantiated multiple times for parallelization, whereas each subtask processes a specific fraction of the data. As Nephele is designed for a Cloud Computing environment, it can decide how many and what types of virtual machines (called instances and instance types respectively) are necessary for job execution. Nephele
also decides what and how many subtasks are running on which instance. The user can influence Nephele’s transformation decisions by putting annotations into the JobGraph. For example the user can specify into how many parallel subtasks a task should be split.

For job execution, the ExecutionGraph gets submitted to Nephele’s runtime system. The runtime system consists of the JobManager, multiple TaskManagers (TMs) and a persistent storage. The JobManager schedules and monitors the execution of the job, whereas the TMs are executing the user defined tasks. If Nephele runs in a Cloud, the TMs are running in virtual machines which Nephele can start and stop automatically. TMs are either running in an own virtual or physical machines or share one with other TMs. The persistent storage is used to store intermediate results if necessary. It is also used for reading the input and writing the final result.

The tasks itself are sending data over so called channels to each other. Channels are data transport mechanisms within the system. Each edge of the JobGraph gets mapped to a channel type in the ExecutionGraph. Nephele supports three types of channels: in-memory-, network- and file-channels. In-memory-channels are using shared memory for data exchange. This is the fastest way for sending data from task to task and it is a pipelined communication. However, this channel can just be used for communication within the same environment. Network-channels are sending the data via a TCP connection to the next task. This is also a pipelined communication, but it enables communication of the tasks in a network environment. File-channels are in opposite to in-memory- and network-channel non-pipelined. If a file-channel gets used, all output data from the first task is written into a temporary file in the persistent storage. After the first task terminated, the second task starts reading the data from this temporary file. As the files are stored in a distributed file system (DFS), file-channels (like network-channels) enable global communication of all tasks with each other.

4.2 Where to collect statistics

The previous section gave a rough overview of Nephele. From a statistic tracking point of view, the communication between the tasks is the most interesting part. This section will therefore give a more detailed look into the design of this communication. First to mention is the basic unit of information within Nephele what is a record. A record might be any piece of information (e.g. a tuple from a relations database, just a number or a string, or maybe something nested like a XML fragment). It is the user’s responsibility to handle the records in the right manner within the tasks. Nephele does not know anything about the internal structure of a record. It just sends a record as a block of bytes from task to task.

We also saw that the JobGraph’s edges get mapped to a channel type. At the time the user is implementing the task, it is not known what channel type will be used later on. Therefore a level of abstraction is given to the user: gates. A gate is hiding the communication details from the user and provides a unique interface for reading (input gate) or writing (output gate) records. The user implemented tasks are using RecordReaders\footnote{This corresponds to the master-worker pattern.}.
and RecordWriters which are reading from or writing into a gate (i.e. it is a one-to-one relationship). Under the hood each gate holds at least one channel which performs the actual communication. Because a task might be executed in many parallel subtasks, each gate can have multiple associated channels (i.e. it is a one-to-many relationship). Each subtask gets a fraction of the input for processing. The following task must therefore receive result records from multiple subtasks. Gates are hiding this from the user, who can write the program as no parallelization would happen. A quick example shall illustrate the relationships of gates and channels. Imagine the following situation: the Cartesian Product of two tables $A$ and $B$ with tuples $a_1, a_2, a_3, a_4$ and $b_1, b_2, b_3, b_4$ respectively should be calculated. The JobGraph consists of four tasks: readA, readB, CP (Cartesian Product) and writeResult. The corresponding JobGraph is shown in figure 4.1. Imagine Nephele decides to split readA, readB and CP into two subtasks each. readA(1) would read $a_1, a_2$, readA(2) would read $a_3, a_4$, readB(1) would read $b_1, b_2$ and readB(2) would read $b_3, b_4$. As CP is executed in two subtasks as well, each CP subtask can handle one output from readB. But both need the hole input from $A$. The corresponding graph (which is similar to an ExecutionGraph) is shown in figure 4.2.

Nephele’s design offers three different points where statistics can be collected: within channels, gates or the RecordReaders/Writers. The advantages and disadvantages of the three possibilities are discussed now.

**Statistic collection within channels**

Collecting statistic within Nephele’s channels is the most granular approach. The advantage of this approach is the possibility to distinguish between different types of channels and its different characteristics. For example, different channel types have different types and sizes of buffers. Collecting statistics within the channels would make it very easy to
writeResult

Figure 4.2: Example of a possible (partial) ExecutionGraph for the JobGraph of 4.1.
monitor buffer sizes and occupations. On the other hand it would be necessary to combine the gathered statistics of all channels belonging to one gate. The channel statistics are too fine granular and not useful for the optimizer. Combining statistics is a non-trivial problem if one thinks on equi-height histograms.

**Statistic collection within gates**

Collecting statistic within gates does not have to solve the problem of combining partial statistics of the channels into one statistic. But a task could get split into multiple subtasks and no statistics about the task itself are available automatically. But it might still be sufficient for the optimizer to use the statistics on subtasks. For the case that a task does not get parallelized the problem disappears anyway. The disadvantage is the fact, that a gate does not have a buffer. The ‘buffer size’ of a gate is therefore the ‘sum’ of the buffer sizes of the underlying channels. As the channels have different types and sizes of buffers it is not obvious what the buffer size of a gate is, or how to calculate the buffer occupation.

**Statistic collection within RecordReaders/RecordWriters**

RecordReaders/RecordWriters are part of the task as well as the gates. Therefore they cannot solve the task parallelization problem either. The problem is that they are instantiated by the user. Therefore it is difficult for Nephele to put a statistic collector within a RecordReader/RecordWriter as Nephele has no access to them. The next disadvantage is the interleaving of task implementation and statistic tracking. It is much better for the user, to focus on the logic of the task while implementing it, without thinking about statistic tracking.

### 4.3 Interface prototyping within Nephele

As pointed out in the section before there are three different points where statistics could be collected. For the interface prototype implementation, gates got chosen. The main reason to reject an implementation within RecordReaders/Writers was to avoid the interleaving of task implementation and statistic tracking. At the same time gates have the advantage over channels to simplify statistic tracking as combining statistics is not necessary so often or can be avoided at all in some cases.

As not each task needs to track statistics, statistic tracking is disabled by default. This avoids any overhead if statistic tracking is not used. The interface provides the possibility to activate statistic tracking when creating the JobGraph at gate level. The user can decide for each specified task what kind of statistics he wants to track on what gate. Nephele will use this information when transforming the JobGraph into an ExecutionGraph and instantiate statistic-tracking-gates instead of regular (non-statistic-tracking) gates. Statistics can be collected on both types of gates (input as well as output). The interface is also flexible enough to track multiple statistics on the same gate. This is important as data-statistics and system-statistic might need to be collected at the same time.
It’s the user’s responsibility to provide a statistic tracking class. As Nephele does not know anything about the records it is processing, it is difficult to provide general statistic tracking classes. However the prototype offers a simple record counter. It might also be possible to implement some generic tracking classes, but this was beyond the scope of this work.

For test purposes, one additional statistic tracking classes got implemented: an equi-width histogram for integers. The record-counter was used in the test as well, as it is the simplest statistic available. The equi-width histogram was used to put a bigger tracking overhead on the system, while it is still fairly simple to implement.

### 4.4 Performance evaluation

To measure the runtime overhead of the interface, five different tests got executed. Each test uses the same JobGraph on the same data. Test 1 performs the job without any statistic tracking activated. Test 2 just activates statistic tracking, but does not use any statistic tracker. The aim was to measure the pure overhead of the interface itself. Test 3 is similar to test 2, but instead of using no statistic tracker, an ‘empty’ statistic tracker got used\(^2\). Test 4 and 5 are tracking the cardinalities of intermediate results (record counting) and building equi-width histograms on the data respectively. The JobGraph which was used for the test is shown in figure 4.3.

Both input tasks read a file of 7.04 GB \((10^9\text{ records})\) line by line and convert each line (ASCII-string) into an IntegerRecord. The main task compares the IntegerRecords (line by line) and calculates the maximum of both for each line. The output task writes the

\(^2\text{Empty means, that the tracking function is empty, i.e. an additional function call per record is measured.}\)
result (7.30 GB – $10^9$ records) on disk again. The difference of the input and output file size is the result of saving the integers as ASCII-strings and not in binary form.

**Setup and configuration**

The test got executed on a Lenovo ThinkPad T410s, with an Intel i5 processor (2.4 GHz), 4 GB main memory and a solid state hard disk (file system: ext4). Nephele was set up within Ubuntu 9.10 (64bit; kernel 2.6.32) using Eclipse SDK 3.5.1 and Sun Java 1.6.0_20. Additionally Hadoop 0.19.2 was used to provide a distributed file system on top of the native ext4 file system of Linux.

Nephele was configured such that each task got executed as one subtask (thread) within one single TaskManager. As all subtasks got executed within one TaskManager, each gate got one InMemoryChannel assigned.

The job runtime was measured within Nephele using time stamps. As time stamps are not very exact (especially in a multi-threaded environment) each test got executed 15 times and the runtime average was calculated\(^3\). As the runtime overhead of tracking statistic is fairly small, each statistic tracker (the empty one, the record counter and the histogram builder) was assigned to each input and output gate (i.e. 6 statistic tracker got used). This ensured that the measured overhead is comparable to the basic run\(^4\).

**Evaluation**

The average runtimes of the tests are shown in figure 4.4. The complete table with all figures is included in appendix A. The runtime overhead show in 4.4 has to be divided by 6 to get the runtime overhead of a single statistic tracker. This results in an overhead of less

---

\(^3\)The big skew of runtime measuring with time stamps was also the reason for using a quite big input to get a long execution time, what lowers this skew.

\(^4\)‘Comparable’ means that the average overhead is about the same size as the standard derivation of the single runs of the base test (test 1).
than 1% for building an equi-width histogram (and even less for record counting). This overhead is quite small what makes statistic tracking useful for real applications. If one put into account that the performed tasks of the job graph are quite simple, the runtime overhead might be even smaller for more complex tasks. The overhead of the interface itself is almost not measurable. The overhead of 0.64% (Test 2) is too small compared to the standard derivation of the base run (1.31% – see appendix A). This overhead could also be a skew in the measured figures.

The overhead difference of 1.98% (Test 3) and 3.05% (Test 4) is quite big for simple record counting. The reason for that is the way statistic tracking got implemented. The aim was to get good structured code instead of high performance code. Therefore one can expect that the overhead could be decreased by one percent point due to a more efficient implementation.
Chapter 5

Conclusion

In this work a statistic tracking interface was defined, to gather statistic over data during execution of a dataflow program. The aim was to collect data which are useful for Adaptive Query Processing (AQP) to improve the execution of the dataflow program. In order to design the interface appropriate, different AQP techniques got discussed. At the moment, it is not clear what AQP techniques will be used later on. Therefore the interface must be flexible enough to support any AQP technique. At the same time it was important, that the runtime overhead for statistic tracking in minimal. A big runtime overhead would make the interface useless. But the performance tests of the implemented prototype showed that the runtime overhead is small enough for practical use. Actually there is still room for decreasing the runtime overhead at a later point (e.g. when the AQP techniques are known and the interface can get optimized for it).

The design of the interface could also be simplified in the following way: as the runtime overhead of interface itself is very small, it might be a good idea, to drop the distinction between regular (no-statistic-tracking) gates and statistic-tracking-gates in order to simplify the code.

Right now the interface just supports monitoring techniques. But it might also be a good idea to use sampling techniques to gather statistics. As sampling spans a very wide research field, a discussion about it was beyond the scope of this work. But it might be interesting for a future work.
## Appendix A

### Performance evaluation table

<table>
<thead>
<tr>
<th>Run</th>
<th>Test one (no statistics)</th>
<th>Test two (no collectors)</th>
<th>Test three (empty collectors)</th>
<th>Test four (record counters)</th>
<th>Test five (histograms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>760,931</td>
<td>749,564</td>
<td>753,104</td>
<td>772,917</td>
<td>783,067</td>
</tr>
<tr>
<td>2</td>
<td>748,327</td>
<td>740,134</td>
<td>748,883</td>
<td>756,497</td>
<td>780,377</td>
</tr>
<tr>
<td>3</td>
<td>746,211</td>
<td>746,825</td>
<td>774,569</td>
<td>779,817</td>
<td>805,522</td>
</tr>
<tr>
<td>4</td>
<td>734,216</td>
<td>755,128</td>
<td>764,518</td>
<td>775,818</td>
<td>796,051</td>
</tr>
<tr>
<td>5</td>
<td>744,956</td>
<td>734,351</td>
<td>765,147</td>
<td>784,034</td>
<td>777,037</td>
</tr>
<tr>
<td>6</td>
<td>732,379</td>
<td>771,623</td>
<td>784,772</td>
<td>756,356</td>
<td>820,165</td>
</tr>
<tr>
<td>7</td>
<td>763,803</td>
<td>758,297</td>
<td>759,235</td>
<td>820,650</td>
<td>788,337</td>
</tr>
<tr>
<td>8</td>
<td>751,051</td>
<td>764,861</td>
<td>767,575</td>
<td>764,266</td>
<td>771,471</td>
</tr>
<tr>
<td>9</td>
<td>743,570</td>
<td>742,503</td>
<td>766,261</td>
<td>756,585</td>
<td>796,939</td>
</tr>
<tr>
<td>10</td>
<td>752,305</td>
<td>747,532</td>
<td>763,488</td>
<td>755,512</td>
<td>782,688</td>
</tr>
<tr>
<td>11</td>
<td>738,816</td>
<td>761,894</td>
<td>753,069</td>
<td>765,559</td>
<td>791,126</td>
</tr>
<tr>
<td>12</td>
<td>767,609</td>
<td>750,933</td>
<td>760,950</td>
<td>764,642</td>
<td>798,913</td>
</tr>
<tr>
<td>13</td>
<td>740,847</td>
<td>749,539</td>
<td>759,342</td>
<td>771,059</td>
<td>782,707</td>
</tr>
<tr>
<td>14</td>
<td>744,161</td>
<td>748,912</td>
<td>754,931</td>
<td>771,819</td>
<td>768,408</td>
</tr>
<tr>
<td>15</td>
<td>740,458</td>
<td>759,283</td>
<td>756,051</td>
<td>755,467</td>
<td>771,628</td>
</tr>
<tr>
<td>Average</td>
<td>747,309 (base)</td>
<td>752,092</td>
<td>762k126</td>
<td>770,067</td>
<td>787,629</td>
</tr>
<tr>
<td>% of base</td>
<td>100.64</td>
<td>101.98</td>
<td>103.05</td>
<td>105.40</td>
<td>105.40</td>
</tr>
<tr>
<td>StDev</td>
<td>10.35</td>
<td>9.866</td>
<td>9.166</td>
<td>16.811</td>
<td>14,135</td>
</tr>
<tr>
<td>% of AVG</td>
<td>1.38</td>
<td>1.31</td>
<td>1.20</td>
<td>2.18</td>
<td>1.79</td>
</tr>
</tbody>
</table>

Figure A.1: Job execution times in milliseconds for each run and evaluation.
Bibliography


