Abstract

Join optimization is one of the most challenging tasks in query processing. The performance of joins depends not only on the algebraical/logical query execution plan (QEP), but also on the chosen join algorithms. Static optimization techniques often suffer from outdated or not available statistics on the data. This may result in sub-optimal QEPs and poor query execution times. Adaptive Query Processing (AQP) tackles this problem. The execution of the query gets monitored and the QEP can be re-optimized if plan sub-optimalities are encountered. This work introduces a new AQP technique by applying a competition model to join operations within routing operators. Our technique allows us, to change the join algorithm during query execution on the fly. As this work is part of the Stratosphere research project, we will run in a highly parallelized MapReduce like environment. Therefore, we assume that the input will fit into main-memory completely. Hence, the overhead of running multiply join-operators in parallel should be negligible and we can expect a significant performance advantage compared to other optimization techniques.
Acknowledgments

First, I would like to thank my supervisor Professor Freytag for his constant support during my studies. Without his help I would have missed the great opportunity of working as an intern for IBM in California. I would like to thank Professor Markl from Technische Universität Berlin for taking time out from his busy schedule to serve as my second examiner. A special thanks goes to my tutor Rico Bergmann. He spent a lot of hours discussing ideas and issues of this thesis with me. These discussions were always very productive and helpful. I also want to thank him for his detailed comments on all the drafts and versions of this work—he spent many hours reading and commenting on it.

The greatest thanks goes to my fianceé Marie who has been already supporting me in all circumstances of life for many years. Especially in the final weeks of completing this thesis you had to spare me many times and you have always been insightfully. Finally, I want to say Thanks! to my parents and their constant support during my whole life. You always encouraged me in any situation and your ongoing support made this thesis even possible.

Danksagung


# Contents

1 Introduction 1
1.1 Overview .................................................. 1
1.2 Conventions ................................................. 3

2 Fundamentals 5
2.1 Programming Models ......................................... 5
  2.1.1 MapReduce ................................................. 5
  2.1.2 Parallelization Contracts ................................. 8
2.2 Basic Join Algorithms ......................................... 9
  2.2.1 Nested-Loop-Join .......................................... 9
  2.2.2 Sort-Merge-Join .......................................... 10
  2.2.3 Hash-Join ................................................ 12
  2.2.4 Summary ................................................ 12
2.3 Symmetric Join Algorithms .................................... 13
  2.3.1 Symmetric-Hash-Join ...................................... 14
  2.3.2 Symmetric-Nested-Loop-Join ............................ 15
  2.3.3 Sort-Merge-Join and Merge-Join ........................ 16
  2.3.4 Summary ................................................ 17
2.4 The Eddy Operator ............................................ 17
  2.4.1 Basic idea of the Eddy operator ......................... 18
  2.4.2 Selectivity routing ...................................... 19
  2.4.3 Back-pressure routing ................................... 20
  2.4.4 Conclusion ................................................. 20
2.5 The Competition Model ........................................ 22

3 The SCORE operator 25
3.1 Choosing joins for competitive execution .................... 25
  3.1.1 Bucket-Nested-Loop-Join ............................... 26
  3.1.2 Symmetric-Bucket-Nested-Loop-Join .................... 27
  3.1.3 Summary ................................................ 29
3.2 Competing Joins .............................................. 29
  3.2.1 Competing join execution .............................. 29
  3.2.2 Competition within SCORE .............................. 31
## CONTENTS

3.3 Complementary Joins ........................................... 32  
3.3.1 Merge-Hash-Join ........................................... 33  
3.3.2 Complementary joins within SCORE ............................ 37  
3.4 SCORE’s routing strategies ..................................... 38  
3.4.1 Blocking operators within a SCORE system ..................... 38  
3.4.2 Partition-aware blocking join operators ...................... 39  
3.4.3 Competition of blocking joins ............................... 40  
3.4.4 Advanced competition strategy ............................... 41  
3.5 Summary .................................................... 42  

4 Evaluation 45  
4.1 Implementation ................................................ 45  
4.2 Experiments .................................................. 47  
4.2.1 Symmetric joins ........................................... 47  
4.2.2 Blocking joins ............................................. 51  
4.2.3 Complementary join ....................................... 53  

5 Conclusion 57  
5.1 Summary .................................................... 57  
5.2 Open issues ................................................ 57  

Appendix 59  
A.1 Graphical User Interface of the SCORE prototype ............... 59  
A.2 SNLJ, SBNLJ and their competing execution .................... 60  
A.2.1 Group shut-down factor 1.00 ................................ 60  
A.2.2 Group shut-down factor 0.75 ................................ 61  
A.2.3 Group shut-down factor 0.50 ................................ 62  
A.2.4 Group shut-down factor 0.25 ................................ 63  
A.2.5 Group shut-down factor 0.05 ................................ 64  
A.3 HJ, SMJ and their competing execution ........................ 65  
A.4 SHJ, MJ and complementary MHJ .............................. 66  
A.4.1 Empty result ............................................... 66  
A.4.2 Regular result ............................................. 67
List of Figures

2.1 A Reduce task which performs a join. 7
2.2 Nested-Loop-Join-Algorithm. 10
2.3 Sort-Merge-Join-Algorithm. 11
2.4 Hash-Join-Algorithm. 12
2.5 Join algorithms, their complexity and other details. 13
2.6 Symmetric-Hash-Join-Algorithm. 14
2.7 Different tuple processing orders of a NLJ. 15
2.8 Symmetric-Nested-Loop-Join-Algorithm. 16
2.9 Merge-Join processing relation $R$ and $S$. 17
2.10 The basic design of an Eddy operator. 18
2.11 Example of a table scan vs. an index access. 23

3.1 NJL vs. BNLJ. 26
3.2 Bucket-Nested-Loop-Join-Algorithm. 27
3.3 Symmetric-Bucket-Nested-Loop-Join-Algorithm. 28
3.4 Shutting down $Op_2$ and moving $R$ and $S$ to $Op_1$. 30
3.5 Logical setup and operator linking of an OpGrp within a SCORE system. 31
3.6 Shutting down an OpGrp and replacing it with the winning operator. 32
3.7 The conceptual setup of an MHJ linked to SCORE. 34
3.8 Example of the out-of-order-tuple filter of the OMJ. 35
3.9 Example of the probe operations of the ASHJ. 37
3.10 Example of a partition-aware blocking join. 40
3.11 Two blocking joins linked to SCORE. 41

4.1 Simulating cost-intensive user-defined join-function. 48
4.2 Runtime in ms of delay-loop. 48
4.3 SNLJ, SBNLJ and competitive execution of both with 0 DUs. 49
4.4 SNLJ, SBNLJ and competitive execution of both with 50 DUs. 50
4.5 SNLJ, SBNLJ and competitive execution of both with 0 DUs and quick shut-down. 52
4.6 HJ, SMJ and competing execution of both. 52
4.7 SHJ, MJ and complementary MHJ with empty result. 54
4.8 SHJ, MJ and complementary MHJ with subtracted routing time. 54
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.9</td>
<td>SHJ and complementary MHJ with ooo-tuples.</td>
<td>55</td>
</tr>
<tr>
<td>4.10</td>
<td>SHJ, MJ and complementary MHJ with result.</td>
<td>56</td>
</tr>
<tr>
<td>A.1</td>
<td>Demo GUI monitoring a running SCORE system.</td>
<td>59</td>
</tr>
<tr>
<td>A.2</td>
<td>SNLJ, SBNLJ and their competing execution with shut-down factor 1.00.</td>
<td>60</td>
</tr>
<tr>
<td>A.3</td>
<td>SNLJ, SBNLJ and their competing execution with shut-down factor 0.75.</td>
<td>61</td>
</tr>
<tr>
<td>A.4</td>
<td>SNLJ, SBNLJ and their competing execution with shut-down factor 0.50.</td>
<td>62</td>
</tr>
<tr>
<td>A.5</td>
<td>SNLJ, SBNLJ and their competing execution with shut-down factor 0.25.</td>
<td>63</td>
</tr>
<tr>
<td>A.6</td>
<td>SNLJ, SBNLJ and their competing execution with shut-down factor 0.05.</td>
<td>64</td>
</tr>
<tr>
<td>A.7</td>
<td>HJ, SMJ and their competing execution.</td>
<td>65</td>
</tr>
<tr>
<td>A.8</td>
<td>SHJ, MJ and complementary MHJ with empty result.</td>
<td>66</td>
</tr>
<tr>
<td>A.9</td>
<td>SHJ, MJ and complementary MHJ with regular result.</td>
<td>67</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Overview

This work is part of the Stratosphere research project [BzBP10, WK09, BEH+10, AHM+10]. The aim of Stratosphere is to develop an information management system running on a compute cloud [WK09]. Stratosphere shall overcome the shortcomings of current parallel data processing systems like Dryad [IBY+07] or Hadoop\(^1\) [Fou09]. Dryad is an execution engine for data-parallel applications. There is no programming model on top of it. The user has to code the program to be executed by specifying a data-flow graph representing the program. Additionally, Dryad runs on a computer cluster with all the shortcomings compared to a compute cloud. The programmer needs to know and understand the hardware architecture of the cluster. Thus making programming more difficult. It is also not possible to allocate new (or release unused) resources during job execution. A compute cloud avoids these drawbacks by providing virtualized resources to the program. This approach hides the actual hardware architecture and virtual machines can be started and shut down at any point in time. Hence, it is easier to develop applications and the computer resources can be used more efficiently. Stratosphere’s aim is to build a system running on a compute cloud. It uses an execution engine for data-flow programs called Nephele [WK09]. Nephele is able to exploit a cloud’s flexibility by allocating and deallocating virtual machines at runtime according to the job’s requirements.

The missing programming model from Dryad [IBY+07] could be compensated by putting a MapReduce [DG04] like system on top of it. But the MapReduce model is too restrictive for some tasks. Stratosphere will host many different use cases (e.g. data cleansing) and must support join operations for them. Even though it is possible to calculate a join with a MapReduce system, the programming model does not support a join natively. Hence, it is more difficult for the programmer to express a join. Another drawback is, that the runtime system is not aware that it has to calculate a join and cannot optimize the execution strategy accordingly. Battré et al. introduced Parallelization Contracts (PACTs) [BEH+10]. PACTs are a generalization of the MapReduce programming model including an easy way to express join operations.

\(^1\)Hadoop is an open source implementation of Google’s MapReduce [DG04].
Additionally, adaptive techniques for query processing will be developed for Stratosphere to avoid the problems of static query optimization. In most relational database management systems static query optimization is done cost-based. The optimizer uses statistical information about the data to find the most efficient Query Execution Plan (QEP) according to the underlying cost model [SAC+79]. Cost-based optimization may fail if statistical information is not up-to-date or missing. As we expect to have almost no statistical information about the data, static cost-based optimization is not applicable within Stratosphere.

Adaptive Query Processing (AQP) tackles the problem of missing statistics by collecting information at runtime. This information can be used to re-optimize the currently running query. A lot of work has been performed on AQP. [GPFS02], [BB05] and [HFC+00] provide a good overview. Most of the work focuses on centralized relational database management systems. Therefore, it is necessary to adopt these techniques for a highly parallel and distributed data-flow system such as Stratosphere.

In this thesis we investigate an adaptive re-optimization technique for join operations. We introduce Stratosphere’s Coordinating Operator for Re-arrangements (SCORE). SCORE is an extended Eddy operator [AH00]. An Eddy is a routing operator which omits System-R [SAC+79] style query optimization completely. Instead of using a QEP it instantiates the needed query operators and routes the tuples (on a per-tuple-basis) through them. The system tries to find the most efficient routing order by monitoring each single operator. The routing approach enables the system to alter the order in which the tuples are sent to the operators at any time and makes the system highly adaptive.

SCORE is enhanced to support a competition model for joins as an additional adaptive optimization technique. Antoshenkov introduced the competition model for base table access within a relational database management system [Ant93a]. The basic idea is to run multiple (partial) QEPs competitive for a short time. While the QEPs are running, the system monitors the performance and decides at runtime which QEP performs best. This QEP is used to calculate the remainder of the query result while all other QEPs are stopped. We extend Antoshenkov’s model by applying it to joins and by integrating the new model into SCORE.

Competing join execution faces new challenges. Join operators have an internal state as they are buffering already processed tuples for later reuse. Therefore, the operators’ performance changes over time. This change makes it more difficult to decide which join algorithm performs better. One idea to solve this problem is to run both algorithms until the first one finishes. As both competing join operators compute the same logical join, we have to ensure that no duplicates will be produced. Therefore, we use a data partitioning strategy for SCORE which avoids duplicates automatically.

SCORE is also able to deal with complementary join algorithms. We developed an adaptive complementary join algorithm called Merge-Hash-Join. It consists of a Merge-and a Hash-Join-Algorithm whereas each calculates a specific part of the result. The idea is to apply a Merge-Join to an almost sorted input. All tuples of the input which are not in-order cannot be processed by the Merge-Join. We use a Hash-Join to process these tuples in order to calculate the missing part of the result.
1.2. CONVENTIONS

Stratosphere will host some very data intensive use cases including the calculation of climate models and text mining. Especially text mining uses cost-intensive user defined join functions. That means that the join is not a simple equi-join for example. We introduce the Bucket-Nested-Loop-Join algorithm in this work. It is a novel approach to improve the performance of a Nested-Loop-Join if a cost-intensive user defined join function is used.\textsuperscript{2}

1.2 Conventions

We are using the following conventions in this work:

- We use capital letters starting with $R$ for relations and input streams (e.g. $R$, $S$, $T$ etc.). We do not distinguish between relations and input streams because we access relations just via input streams.

- The number of tuples in a relation $R$ is referenced as $|R|$.

- If two relations are joined, the join-result is represented as the concatenation of the two relation symbols. The result relation for a join of $R$ and $S$ will be $RS$ for example.

- We use lower case letters for tuples. The letter corresponds to the relation the tuple belongs to, i.e. a tuple $r$ belongs to relation $R$.

- If two tuples $r$ and $s$ are joined the result tuple is $\langle r, s \rangle$.

- We use upper case letters (different to $R$, $S$, $T$ etc.) with an index for operators. The index is a relation symbol the operator is defined on. A filter operator which is defined on $R$-tuples for example is $F_R$. We do not specify the actual attribute of $R$ on which $F_R$ is defined on, because this detail is not important in our discussion.

- $h$ is used for a hash table and $h(r)$ is the bucket belonging to $r$ (i.e. the result of the probe operation from $r$ on $h$). The attributes from $r$, the hash value is calculated on, is usually not mentioned explicitly, as it is clear out of the context.

- We assume for any hash table, that each bucket contains values of the same key explicitly, i.e. for any two keys $k_1, k_2$: $h(k_1) \neq h(k_2) \Leftrightarrow k_1 \neq k_2$. This assumption is different to the usual definition of a hash table. We use this modified definition of a hash table because it simplifies the description of some algorithms.\textsuperscript{3}

\textsuperscript{2}User defined join functions forces the usage of a Nested-Loop-Join because nothing is know about the function. This missing knowledge prohibits the usage of merge or hash strategies.

\textsuperscript{3}This modified assumption is no limitation and all pseudo-code examples can be altered to be correct using a standard hash table.
Chapter 2

Fundamentals

2.1 Programming Models

We discuss the details of the MapReduce [DG04] programming model and Parallelization Contracts (PACT) [BEH+10, AHM+10] in this section. Parallelization Contracts got developed for Stratosphere because the MapReduce model is too restrictive due to the missing native support of joins. It might be possible that new PACTs will be added to the model during the research work on Stratosphere.

2.1.1 MapReduce

The MapReduce programming model was introduced by Dean and Ghemawat [DG04]. It was developed to simplify the programming of parallel applications. The idea is to hide the parallelization overhead from the user. The programmer can still write sequential code and the execution engine takes care of parallelizing it during execution. To enable the engine to parallelize the program, it must follow some restrictions. MapReduce is inspired by functional programming and we introduce first- and second-order functions which are a basic concept in functional programming [Pep00].

Definition 2.1 A first-order function is a function taking data values as input and outputs a data value. Higher-order functions in general are functions which have other functions (of lower order) as their input or output (next to the data values). A second-order function is a function having a first-order function as input or output.

The idea of first- and second-order functions is that the second-order function applies a first-order function to input data. In the MapReduce model the user has to implement first-order functions that contain the actual semantics of the program. The MapReduce programming model provides two second-order functions—Map and Reduce—which apply the user-defined first-order functions to input data.

MapReduce uses the key/value-pair data model. In this model a key is not a unique value over all tuples. It is an identifier and is assigned by the Map function. The key is used by the system for sorting and grouping. The value of a key/value-pair is an arbitrary (and maybe complex) type which is not exposed to the system. It is just accessed by the user-defined first-order functions.
We are now explaining the detailed semantics of the two second-order functions which are provided by the MapReduce programming model.

**Map**  
The second-order Map function takes a user-defined function and a set of key/value pairs as input. We refer to the user-defined function as `map` (with lower case letter) in the remainder of this work. Map applies the given function to each key/value pair in an arbitrary order. Hence, `map` takes a single key/value pair as input. While processing the input `map` can emit zero, one or more key/value pairs. It can modify the key or the value or both of them. A simple example of a `map` function is the implementation of a filter operation. In this case, `map` does not modify the value\(^1\) and emits the key/value pairs which satisfy the filter condition. The Map function applies the given `map` to each key/value pair exactly once. The output of a Map step consists of all emitted tuples from all `map` calls. This intermediate result is passed to a Reduce function as input.

**Reduce**  
Before the Reduce step can be executed, the intermediate result of the Map step is partitioned on the keys of the key/value pairs. Reduce takes a user-defined function (`reduce`) as well as the set of data partitions as input. Reduce applies the user-defined function to each data partition exactly once. It does not guarantee an order in which the partitions will be processed. The given `reduce` function takes a single key and a list of all values of the according data partition as input. The result of `reduce` is a list of values which can be empty. In most practical cases `reduce` returns just one value because it calculates an aggregation of the input values (e.g. sum, avg etc.). The complete Reduce step is comparable to the `GROUP BY` SQL clause with aggregation.

A MapReduce job consists of a single Map step followed by a single Reduce step. For more complex operations, multiple MapReduce iterations are necessary. Google’s MapReduce implementation [DG04] uses its own distribution file system GFS (Google-File-System) [GGL03] for its MapReduce implementation. All intermediate results (i.e. after each Map and Reduce step) are written back to the file-system. Persisting intermediate results makes failure recovery easier as these check-points\(^2\) are written to disk regularly.

A Map step can be executed in a highly parallel manner. As each `map` call is independent, many `map` tasks can run in parallel. The system can give each `map` task an arbitrary part of the input data without violating the semantics. The implementation from Dean and Ghemawat [DG04] splits the complete input into \(M\) pieces of equal size where \(M\) is a user-defined parameter (e.g. \(M\) is the number of available machines). The output of the `map` tasks gets partitioned into \(R\) splits (\(R\) is provided by the user). After partitioning the key/value pairs by the key, a `reduce` task is started for each partition once. As the key/value pairs are partitioned already, these calls are independent from each other enabling the system to run many parallel `reduce` tasks.

---

\(^1\)If the key needs to be modified depends on the following Reduce task and is a semantic question linked to the use case.

\(^2\)A check-point is a point where all data (i.e. intermediate result) is written to disk. These check-points enable the system to continue working from these points on after a crash and avoid re-calculating already processed steps.
Some problems fit natively into the MapReduce model (e.g., distributed grep [DG04]). But there are also many problems which cannot be modeled easily. The most important example in database systems is a join operation. One reason is that neither Map nor Reduce supports multiple inputs. It is still possible to implement a join within a MapReduce system and some work was done to improve the performance of joins within a MapReduce system [AU10]. But the semantic of the join is not known by the runtime system because it is hidden within the user-defined functions. An equi-join\(^3\) for example can be implemented using two Map and one Reduce task. Each Map task reads one input relation and sets the key of the output pairs to be the join-attributes. Furthermore, it adds the name of the input relation to the data-value. The Map-output gets partitioned on the key, i.e., on the join-attributes. Hence, all values (from both relations) with the same join-attributes are used for a single reduce call. The user-defined reduce function groups the input by the relation-name of each value. Than it outputs the Cartesian Product calculated on both groups. Figure 2.1 illustrates a Reduce task joining two relations \( R \) and \( S \). The figure shows a single reduce call on one data partition containing all \( R \)- and \( S \)-tuples with key \( k_i \). The input on the bottom is the remaining part of not processed input tuples for this call. The next input tuple \( s_5 \) belongs to relation \( S \). It will be added to the \( S \)-group and joined with all tuple from the \( R \)-group \( (r_1, r_2, r_3) \).

\(^3\)An equi-join uses ‘\( = \)’ as join-predicate.
CHAPTER 2. FUNDAMENTALS

The MapReduce runtime system provides a list containing \( S \)- and \( R \)-tuples with the same key for the \textit{reduce} function. The user code must distinguish the tuples and buffer them internally to calculate the join result. Therefore, the user has to add the name of the relation, a tuple comes from to the value within the \textit{map} task. This meta information needs to be removed by the \textit{reduce} task later on. A native join support does not require the user code to add meta information to the data. Such a system is able to distinguish the tuples automatically by providing a second-order function supporting two data-inputs.

2.1.2 Parallelization Contracts

\textit{Parallelization Contracts} (PACTs) [BEH\textsuperscript{+}10, AHM\textsuperscript{+}10] are a novel approach to overcome the shortcomings of the MapReduce model. PACTs generalize the MapReduce model by introducing \textit{input-contracts}. Input-contracts are second-order functions taking a user-defined first-order function as input. The PACT model defines a \textit{Map} and a \textit{Reduce} input-contract which are equivalents to the Map and Reduce tasks of the MapReduce model respectively. In addition to the \textit{Map}- and \textit{Reduce}-contract, there are input-contracts which accept multiple inputs: \textit{Cross}, \textit{CoGroup} and \textit{Match}. These contracts can be used to implement the Cartesian Product as well as outer- or inner-join respectively. The \textit{Cross}-contract for example applies the user-defined function to all key/value pairs which can be build over the two inputs. The \textit{Match}-contract calls the user-task for all key/value pairs over the two inputs which have the same key—it can be used to express an inner equi-join.

The execution of the input-contracts can be parallelized similarly to the parallelization strategy of a MapReduce system. The contracts are defined in such a way that each call of the user-defined function is independent. Hence, many parallel execution threads can be started. The contracts with two data-inputs are more difficult to parallelize. The \textit{Cross}-contract for example involves data-replication strategies because a single key/value pair from one input matches with all key/value pairs of the other input. The system distributes this key/value tuples over different machines to execute the single user-function-calls in them. Thus, one input is partitioned over all machines while the other is replicated for each machine. Another main difference to the MapReduce system is, that the intermediate result of each step is not written to disk automatically. Avoiding check-points reduces cost intensive I/O-operations and speeds up performance. The drawback of this strategy is that failure recovery might become more expensive. Check-points are not written to disk as regularly and re-execution of already processed steps might be necessary.

Next to the input-contracts there are \textit{output-contracts}. Output-contracts are optional annotations for the user-defined first-order functions. They contain information about the semantics of the user-defined function. Output-contracts enable the system to find a more efficient execution strategy. For example there is the \textit{same-key}-contract which tells the system that the function will not modify the key. In case of a \textit{Reduce}-PACT, where the input gets sorted before hand, the system knows that the sorting order is preserved and an additional sorting step can maybe be omitted. Other output-contracts are \textit{super-key}, \textit{unique-key} and \textit{partitioned-by-key} [BEH\textsuperscript{+}10].

As the PACT model provides second-order functions which have a join semantic, the runtime system can use join algorithms to improve the execution strategy. Many different join strategies are known and it is an optimization problem to find the best performing
2.2. BASIC JOIN ALGORITHMS

2.2 Basic Join Algorithms

In this chapter, we discuss the pros and cons of different join algorithms. We have to decide what algorithms we would like to use in Stratosphere, and what algorithms we can run competitive. In the pseudo-code examples, \( R \) and \( S \) are the two input relations which are joined. The function \( JOIN(\ldots) \) returns \( true \) if the two input tuples satisfy the join predicate—\( false \) otherwise.

We compare the worst case time (\( C_T \)) and space (\( C_S \)) complexities of the algorithms with one limitation. The result of a join could always be the Cartesian Product of the input relations what has quadratic size. Hence, the worst case time complexity is quadratic for all joins. Instead we compare the number of comparisons (join-predicate-evaluations) an algorithm needs to calculate the result. The theoretical best case complexity is \( O(|R| + |S|) \), as we cannot skip any tuple without looking at it.

If we compare the space complexity, we do not look at the output either for the same reason. The space complexity is the needed main memory\(^5\) for buffering tuples within the algorithm. We assume that both inputs are just accessible via a \( \text{next()} \) interface. \( \text{next()} \) returns the next tuple from the input or \( \text{null} \) if no more tuples are available. If an algorithm needs to access a tuple more than once it has to buffer it internally. This internal buffer size determines the space complexity of the algorithm.

We start with some basic join algorithms which are relevant for the Stratosphere project. There are a few other join algorithms in centralized database management systems which are not applicable. One example is an Index-Join. Stratosphere will not support indices in the first place and hence, we cannot use an Index-Join. The following three algorithms are also described in Silberschatz’s book ‘Database Systems Concepts’ [SKS98].

2.2.1 Nested-Loop-Join

The Nested-Loop-Join (NLJ) is the simplest join algorithm. It takes every tuple from both inputs and evaluates the join condition for all tuple pairs. For each tuple of the first (outer) relation a loop over all tuples of the second (inner) relation is performed. Each step of the loop tests if the join condition is satisfied (figure 2.2).

The NLJ has quadratic time complexity (\( C_T \in O(|R| \cdot |S|) \)) and can evaluate any join predicate. There are other join algorithms with a lower complexity—but these algorithms are usually limited to perform equi-joins (i.e. \( JOIN(r,s) := r.p = s.q \), where \( p,q \) are the join attributes).

The space complexity is \( C_S \in O(|S| + 1) \) what is \( O(|S|) \): the algorithm buffers the inner relation \( S \) in memory while it scans the outer relation once.

---

\(^4\)The integration of SCORE into Stratosphere is beyond the scope of this thesis.

\(^5\)In the context of Stratosphere we assume that the input fits into main memory.
for each tuple \( r \) in \( R \) do
  for each tuple \( s \) in \( S \) do
    if \( \text{JOIN}(r,s) = \text{true} \) then
      output \( \langle r, s \rangle \)
    endif
  endfor
endfor

Figure 2.2: Nested-Loop-Join-Algorithm.

### 2.2.2 Sort-Merge-Join

The idea of the Sort-Merge-Join-Algorithm (SMJ) is to reduce the number of comparisons by sorting both input relations before joining them. Sorting can only be done if there is a total order on the join attributes. The \( \text{sort}(\cdot) \) function in the pseudo-code example (figure 2.3) sorts relations \( R \) and \( S \) on their join attributes \( p \) and \( q \) respectively. Additionally, the join must be an equi-join.\(^6\) If both requirements are satisfied, the algorithm works as follows: both sorted inputs are read alternating until the join-predicate is satisfied, i.e. \( r.p = s.q \). The algorithm reads from the input which tuples are currently smaller compared to the other input. If a match is found, the Cartesian Product of all consecutive tuples which are equal (of both inputs) is outputted as part of the result.

The complexity of the actual join-part (lines 28–36) has linear runtime because we do not count the output step (lines 31–35). It remains the sorting overhead (lines 13/14) what is \( O(n \log n) \) where \( n \) is the number of tuples to be sorted, if we use the Merge-Sort- or Quick-Sort-Algorithm [Knu98]. Hence, the overall complexity of SMJ is \( O(|R| \cdot \log |R| + |S| \cdot \log |S| + |R| + |S|) \). This term is dominated by sorting the bigger input relation. Let

\[
Z := \begin{cases} 
R, & \text{if } |R| > |S| \\
S, & \text{otherwise}
\end{cases}
\]

than \( C_T \in O(|Z| \cdot \log |Z|) \).

There are cases where the input relations are sorted on the join predicates beforehand. The sorting part of the SMJ algorithm than can be skipped. We refer to this variation of the SMJ algorithm as Merge-Join (MJ) algorithm. As it omits the sorting step which dominates the runtime of the SMJ, the time complexity of MJ is \( C_T \in O(|R| + |S|) \). Each tuple is compared a maximum of three times (lines 20, 24, 28) but the constant 3 can be omitted in the \( O \)-notation. The MJ’s time complexity meets the theoretical best case complexity for any join algorithm.

The space complexity \( C_S \) of SMJ and MJ is \( O(|R| + |S|) \). SMJ needs to buffer both sorted relations \( R' \) and \( S' \). But \( O(|R| + |S|) \) is also needed in lines 29/30 if all tuples are equal on the join attributes. In this case \( L_r \) and \( L_s \) are \( R' \) and \( S' \) respectively. But the Cartesian Product as a result is very uncommon in practice. In the (opposite and

\(^6\)It is also possible to modify the algorithm to perform a \( \Theta \)-join with \( \Theta \in \{<; \leq; \geq; >\} \).
2.2. BASIC JOIN ALGORITHMS

function getAll(t,X)
L := \{t\}
t' := next(X)

while t'.a = t.a do /* where a is the join attribute of t */
    L := L \cup \{t'\}
t' := next(X)
endwhile

/* t' is the first tuple which is not equal to t */
return with \langle t', L \rangle

R' := sort(R)
S' := sort(S)
r := next(R')
s := next(S')

while r \neq null and s \neq null do
    while r.p \lt s.q do
        r := next(R')
    endwhile
    while s.q \lt r.p do
        s := next(S')
    endwhile
    if r.p = s.q then
        \langle r, L_r \rangle := getAll(r, R')
        \langle s, L_s \rangle := getAll(s, S')
        for each tuple r' in L_r do
            for each tuple s' in L_s do
                output \langle r', s' \rangle
            endfor
        endfor
    endif
endwhile

Figure 2.3: Sort-Merge-Join-Algorithm.
for each tuple $r$ in $R$ do
  $h(r) := h(r) \cup \{r\}$
endfor

for each tuple $s$ in $S$ do
  for each $r'$ in $h(s)$ do
    output $\langle r', s \rangle$
  endfor
endfor

Figure 2.4: Hash-Join-Algorithm.

very common) case of unique join attribute values, $C_S$ is $O(1)$ for lines 29/30 and hence $C_S \in O(1)$ for MJ (not for SMJ because of the sorting space requirements).

2.2.3 Hash-Join

The Hash-Join-Algorithm (HJ) reduces the overhead by hashing one relation on the join attributes. Building a hash table can be done in linear time with an appropriate hash function. The HJ-Algorithm hashes just one relation and does not require an order on the join attributes. After the hash table is build, the second relation is scanned, and each tuple is probed into the hash table to find join partners. Figure 2.4 shows a pseudo-code example of the HJ.

Recall that we assume that each bucket of a hash table contains values with the same key explicitly. Otherwise we could not perform the loop in line 6 without a key comparison. Our unique-bucket-requirement increases the complexity of finding the right bucket for a given key (this in enclosed in $h(.)$). If the hash value of two distinct keys is the same, we need to find an alternative bucket for those keys. One idea is to re-hash the key until the right bucket is found. Another strategy is to keep a list of all buckets for all keys having the same hash value. In this case, a linear or binary search over the list is done in order to find the bucket belonging to the given key. In either case (even if we drop the unique-bucket-requirement and do a key comparison in the probe phase) the complexity is $O(|R| \cdot |S|)$.

In practice, however, we can assume a time complexity of $C_T \in O(|R| + |S|)$. Even if some hash collisions\(^7\) occur, they can be resolved in constant time. Hence, the hash part is linear in $|R|$ and the probe part is linear in $|S|$\(^8\). Thus, the HJ has the best possible time complexity. The space complexity of HJ is linear in $|R|$ because all $R$-tuples are buffered within $h$ but $S$ is scanned just once.

2.2.4 Summary

Figure 2.5 shows a table summarizing the differences of the discussed join algorithms. The table includes time and space complexity as well as other limitations and requirements. HJ and MJ have the best possible time complexity. But SMJ, MJ and HJ have the limitation

---

\(^7\)A hash collision occurs if two distinct keys have the same hash value.
\(^8\)Once again, we count comparisons ($h(.)$ calls) only.
2.3. SYMMETRIC JOIN ALGORITHMS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time complexity</th>
<th>Space complexity</th>
<th>(\Theta)-Join</th>
<th>requires</th>
<th>symmetric version</th>
</tr>
</thead>
<tbody>
<tr>
<td>NLJ</td>
<td>(</td>
<td>R</td>
<td>\cdot</td>
<td>S</td>
<td>)</td>
</tr>
<tr>
<td>SMJ</td>
<td>(</td>
<td>Z</td>
<td>\cdot \log</td>
<td>Z</td>
<td>) (^1)</td>
</tr>
<tr>
<td>MJ</td>
<td>(</td>
<td>R</td>
<td>+</td>
<td>S</td>
<td>)</td>
</tr>
<tr>
<td>HJ</td>
<td>(</td>
<td>R</td>
<td>+</td>
<td>S</td>
<td>)</td>
</tr>
</tbody>
</table>

\(^1\) Where \(Z := \begin{cases} R, & \text{if } |R| > |S| \\ S, & \text{otherwise} \end{cases} \)

\(^2\) \(<, \leq, \geq, >\) are possible to implement, but not common.

Figure 2.5: Join algorithms, their complexity and other details.

to work on equi-joins only. That is the reason why the NLJ with its worse time complexity is still important: it can evaluate any join condition. The requirement of a total order for the SMJ is not severe in practice as the most common data types like numbers, dates, or strings have a total order.

The last column indicates whether there is a symmetric version of the algorithm or not. Symmetric join algorithms are the topic of the next section and are explained there.

2.3 Symmetric Join Algorithms

The discussed join algorithms of the previous section have a fixed order in which they process the input. The HJ for example first builds the hash table on input \(R\) before the probing phase begins. During this time no output is produced. This initial delay is critical for a data-flow environment [WA91]. All operators consuming the output of the HJ have to wait until the build phase is finished. Operators like the HJ are called blocking operators because they consume one or both relations entirely before emitting results. The opposite of blocking operators are pipeline operators. Those operators do not have a preparation phase and output partial results from the beginning on. Additionally to the regular delay of the preparation phase, it could happen that the input stream of relation \(R\) delays. In a distributed environment like Stratosphere data is often sent via network from one node to another. Network failures or a crash of a single node (including recovery overhead) are expected to happen. Both can result in a delayed data delivery for a join algorithm. In the case of a HJ a delayed input for the hash relation prevents the HJ to make progress at all, increasing the initial delay. The idea of symmetric join algorithms is to produce output whenever possible. In the case of a HJ and a delayed input stream of relation \(R\) for example, the symmetric version of the HJ (Symmetric-Hash-Join) would start probing available \(S\)-tuples with already hashed \(R\)-tuples. Hence, the algorithm makes progress at any point in time if new input tuples are available. Additionally, it processes \(R\) and \(S\) alternating to avoid initial delays. The details of the Symmetric-Hash-Join are described in subsection 2.3.1.

Symmetric join algorithms are pipelined and can proceed working even if one input delays. They have also the advantage, that they can be implemented multi-threaded. A multi-threaded implementation increases parallelism resulting in a performance advan-
$r := \text{next}(R)$
$s := \text{next}(S)$

while $r \neq \text{EOS}$ or $s \neq \text{EOS}$ do

  if $r = \text{null}$ then
    $r := \text{next}(R)$
  endif

  if $r \neq \text{null}$ and $r \neq \text{EOS}$ then
    $h_R(r) := h_R(r) \cup \{r\}$
    for each $s'$ in $h_S(r)$ do
      output $\langle r, s' \rangle$
    endfor
    $r := \text{next}(R)$
  endif

  if $s = \text{null}$ then
    $s := \text{next}(S)$
  endif

  if $s \neq \text{null}$ and $s \neq \text{EOS}$ then
    $h_S(s) := h_S(s) \cup \{s\}$
    for each $r'$ in $h_R(s)$ do
      output $\langle r', s \rangle$
    endfor
    $s := \text{next}(S)$
  endif
endwhile

Figure 2.6: Symmetric-Hash-Join-Algorithm.

tage. But the symmetric implementation results in more overall work and a higher space complexity compared to their blocking versions. The time complexity might still be the same, but at least the constant factors in the complexity functions are higher.

### 2.3.1 Symmetric-Hash-Join

The Symmetric-Hash-Join (SHJ) was introduced as pipelining hash-join by Wilschut and Apers [WA91]. It processes both inputs symmetrically by building a hash table on both inputs and probing against the other. We introduce an End-of-Stream-tuple (EOS) for the pseudo-code example. An EOS-tuple is no regular data tuple but a marker for the end of the data-stream. The $\text{next}(.)$ function returns EOS if no more tuples are in the stream. If no tuple is available because of a delayed stream, the $\text{next}(.)$ function returns null.

The main loop (lines 4–20, figure 2.6) processes both relations symmetrically. The probe and hash operations are alternating what avoids the initial delay of building a complete hash table like the HJ algorithm does. The time complexity of the SHJ is the same as for the non-symmetric version. Nevertheless more work is done as two hash
2.3. Symmetric Join Algorithms

Tables are built. The SHJ performs about twice as many operations as the HJ. Its space complexity is $C_S = O(|R| + |S|)$ as both relations are buffered in a hash table.

2.3.2 Symmetric-Nested-Loop-Join

The NLJ reads the inner relation multiple times while it scans the outer just once. Hence, most implementations buffer the inner relation in main memory (if possible). If we use this algorithm in a distributed environment and the inner relation delays, the algorithm blocks. Figure 2.7(a) shows the processing order of the NLJ. If the inner relation delays, the first scan of it (top arrow) cannot proceed. The NLJ will not process any other tuples but block. If we consider buffering the inner relation as an initial step, the first output tuple is generated, after the inner relation is buffered completely. But actually it would be possible to join received inner tuples immediately with available outer tuples. The idea of the Symmetric-Nested-Loop-Join (SNLJ) was introduced by Avnur and Hellerstein [AH00] (figure 2.7(b)). They identified so called moments of symmetry within the NLJ where outer and inner relation can be exchanged without violating the semantics of the join. Each time the scan of a relation is completed, the processing order of both can be switched. This idea results in the SNLJ: blocking is avoided by exchanging the inner with the outer relation (at specific points) if the inner delays. Figure 2.7(c) shows the scanning order when the first relation (horizontal—let it be $R$) delays after six tuples. The next tuple from the second relation (vertical—let it be $S$) is taken to join it with all available $R$-tuples. After three $S$-tuples got processed (top three horizontal arrows) the $S$-stream delays, but new $R$-tuples are available. Hence, the processing order is exchanged. The next $R$-tuple is taken and a scan over all buffered $S$-tuples is performed. This scan happens twice (the two vertical arrows). Now $R$ delays again and another switching between inner- and outer-relation happens. A pseudo-code example of the SNLJ is given in figure 2.8.

The time complexity of the SNLJ is the same as for the NLJ. But space complexity doubles as both relations have to be buffered. The SNLJ’s space complexity is $C_S = O(|R| + |S|)$ compared to $C_S = O(|S|)$ for the NLJ.
\[ r := \text{next}(R) \]
\[ s := \text{next}(S) \]

\[ \text{while } r \neq \text{EOS or } s \neq \text{EOS do} \]
\[ \text{if } r = \text{null then} \]
\[ r := \text{next}(R) \]
\[ \text{endif} \]
\[ \text{if } r \neq \text{null and } r \neq \text{EOS then} \]
\[ B_R := B_R \cup \{r\} \]
\[ \text{for each } s' \text{ in } B_S \text{ do} \]
\[ \text{if } \text{JOIN}(r, s') = \text{true then} \]
\[ \text{output } \langle r, s' \rangle \]
\[ \text{endif} \]
\[ \text{endfor} \]
\[ r := \text{next}(R) \]
\[ \text{endif} \]
\[ \text{if } s = \text{null then} \]
\[ s := \text{next}(S) \]
\[ \text{endif} \]
\[ \text{if } s \neq \text{null and } s \neq \text{EOS then} \]
\[ B_S := B_S \cup \{s\} \]
\[ \text{for each } r' \text{ in } B_R \text{ do} \]
\[ \text{if } \text{JOIN}(r', s) = \text{true then} \]
\[ \text{output } \langle r', s \rangle \]
\[ \text{endif} \]
\[ \text{endfor} \]
\[ s := \text{next}(S) \]
\[ \text{endif} \]
\[ \text{endwhile} \]

Figure 2.8: Symmetric-Nested-Loop-Join-Algorithm.

### 2.3.3 Sort-Merge-Join and Merge-Join

It is not possible to implement the SMJ in a symmetric way. The reason for that is the sorting step at the beginning of the algorithm. The algorithm cannot start joining until both sorting steps are completed. Hence, the algorithm blocks until both inputs are available completely.

The MJ on the other hand is symmetric natively. It interleaves reading the left and right input and outputs tuples from the beginning without any preparation phase. A delayed input is no problem either. After a tuple got processed, we know that we have already joined all available tuples with all matching tuples. Figure 2.9 shows an example of the MJ processing relation \( R \) and \( S \) (both sorted). For simplicity we assume to have unique values in both relations. The MJ just finished processing the current tuples \( r \) and \( s \). We know that all previous processed tuples \( (R_{\text{prev}} \lor S_{\text{prev}}) \) are smaller than \( r \) and \( s \) respectively. They are also smaller than \( R_{\text{cont}} \lor S_{\text{cont}} \). Hence, the result of \( R_{\text{prev}} \lor S \)
2.4. THE EDDY OPERATOR

and \( R \bowtie S_{prev} \) is calculated completely. If for example relation \( R \) delays, there is no need to take special action like the SNLJ and SHJ do. The SNLJ and SHJ switch to the other relation and continue processing if one input delays. The MJ on the other hand has already performed all possible tuple comparisons. There is no need to compare available \( S \) tuples (from \( S_{cont} \)) with already processed \( R \)-tuples as no such tuple-pair can satisfy the join condition. Available \( S \)-tuples from \( S_{cont} \) can join with (not available) tuples from \( R_{cont} \) only. There is no work the MJ could perform if one input delays.

2.3.4 Summary

Symmetric join algorithms are pipelined and consume both input relations alternating. They produce output even if one input relation delays. Additionally, initial delays are avoided. Initial delays can be critical in a data-flow system as operators following a blocking operator cannot start working until the (blocking) initial phase finished. The blocking Sort-Merge-Join can still be an important algorithm for our system. The output of the Sort-Merge-Join is sorted. A sorted output could maybe be exploited by the system if following operations perform better on sorted input.

The blocking nature of the Sort-Merge-Join can be weakened by splitting the input into partitions. The Sort-Merge-Join is then started partition-wise. This strategy shortens the initial blocking phase. At the same time there are intermediate blocking phases, each time the algorithms starts to process a new partition. The drawback of this strategy is that the algorithm’s complexity is increased with an increasing number of partitions. That is the reason why we do not refer to this algorithm as Symmetric-Sort-Merge-Join. Additionally, this strategy is not limited to the Sort-Merge-Join but can be applied to all other blocking join algorithms as well. A detailed discussion about this partition-blocking join strategy follows in subsection 3.4.2.

2.4 The Eddy Operator

The *Eddy* operator was introduced by Ron Avnur and Joseph M. Hellerstein [AH00]. It is a routing operator for query execution. The Eddy is connected with operators (like selects or joins) which perform specific query steps. The Eddy routes the tuples to the operators dynamically. The basic Eddy design is shown in figure 2.10. The input streams are putting new tuples into the system. The Eddy decides on a per-tuple basis to which operator it sends a tuple. The single operators are processing the tuples and returning intermediate result tuples to the Eddy. These intermediate result tuples are sent to other
operators until a tuple was processed by all operators. The result tuples are returned finally.

The Eddy approach shall be used within Stratosphere in order to be able to alter the current running data-flow program. Stratosphere has an additional layer of abstraction above the PACT programming model. This layer provides declarative (query) languages. A compiler translates these queries into PACT programs. Hence, there might be multiple different data-flow programs which express the semantics of the original declarative query. If there are multiple Map-contracts—each implementing a filter on one input for example—, the order of these Maps can be altered without violating the semantics. The Eddy approach makes it easy to change the order of different operators within a data-flow program. The integration of the routing approach into Stratosphere is beyond the scope of this work. PACT programs have a different (and generalized) semantic than Query Execution Plans and it is not trivial to apply the Eddy’s approach to a PACT program. In this thesis we extend the Eddy with a competition model for joins. But first we discuss how Avnur’s and Hellerstein’s Eddy works in detail.

2.4.1 Basic idea of the Eddy operator

The Eddy [AH00] was developed for continuous query re-optimization. It omits a static System-R style Query Execution Plan (QEP) [SAC+79]. Instead it can adopt the routing order dynamically at any point in time with regards to the current system conditions.

A short example illustrates how an Eddy works: we assume to have two input streams \( R \) and \( S \), a filter operator for \( R \)-tuples \( (F_R) \) and a \( R-S \)-join operator \( (J_{RS}) \). \( F_R \) returns a tuple to the Eddy if the tuple satisfies the filter predicate. Otherwise the tuple is dropped by \( F_R \). \( J_{RS} \) buffers all \( R \)- and \( S \)-tuples in separate internal buffers. For each pair \( r_i \in R \) and \( s_j \in S \) it tests, if \( r_i \) and \( s_j \) are passing the join condition. If \( r_i \) and \( s_j \) satisfy the join predicate, the joined tuple \( \langle r_i, s_j \rangle \) is returned to the Eddy.

If a new \( R \)-tuple arrives, the Eddy can route it either to \( F_R \) or \( J_{RS} \). The Eddy decides for every tuple individually to which operator the tuple is routed. New arriving \( S \)-tuples can only be sent to \( J_{RS} \). If \( J_{RS} \) returns a \( \langle r, s \rangle \)-tuple, the Eddy sends it to \( F_R \) before it
2.4. THE EDDY OPERATOR

can be emitted. \( F_R \) evaluates the filter predicate on the \( r \) part of the \( \langle r, s \rangle \)-tuple. If the predicate is satisfied, the \( \langle r, s \rangle \)-tuple is returned to the Eddy which sends it to the output. It could also happen that the \( r \) part of a \( \langle r, s \rangle \)-tuple (returned from \( J_{RS} \)) got already evaluated by \( F_R \). In this case, the Eddy outputs the \( \langle r, s \rangle \)-tuple immediately. The \( r \) part could have been already evaluated if the initial \( R \)-tuple was routed to (and returned to the Eddy from) \( F_R \) in the first place.

The Eddy uses status bits assigned to each tuple to keep track to what operators a tuple can be sent to (or was sent to already). There are two bit arrays of equal size: the ready-bits and the done-bits. The size of the arrays depends of the number of operators (and the number of inputs of each). It is the sum of all inputs over all operators. In our example there are \( F_R \) which has one input and \( J_{RS} \) which has two inputs. Hence, the used bit arrays have a size of three\(^9\). If a ready-bit is set, it indicates that the tuple can be sent to the corresponding operator. If the associated done-bit is also set, the tuple was already processed by this operator and must not be sent to it a second time. The Eddy outputs a tuple if all done-bits are set. The initial ready-bits can be set either by the input-streams or by the Eddy, each time a new tuple is sent to it.

Additionally, the Eddy has to have an initial routing order. Avnur and Hellerstein suggested that a pre-optimizer determines this initial order. The initial order must not be calculated with a high precision as it will be adopted by the Eddy anyway. In the example above there are two possible routing orders for \( R \)-tuples (1st: \( F_R - J_{RS} \) and 2nd: \( J_{RS} - F_R \)) and one for \( S \)-tuples (\( S \)-tuples can just be sent do \( J_{RS} \)). The main problem is to decide when and how to adopt the routing order. The aim is to use a most efficient order at any point in time to minimize the time for processing all tuples. The problem is that the Eddy does not know anything about the operators beforehand. It has to monitor the behavior of the operators and adopt the routing order accordingly. Avnur and Hellerstein are suggesting two different routing strategies to achieve this goal as good as possible. Both strategies are discussed in the following two subsections.

2.4.2 Selectivity routing

The first routing strategy is called selectivity routing. The selectivity of an operator is the fraction of returned tuples over all possible output tuples. A filter can possibly return all input tuples. Hence, the selectivity for a filter operator is \( \frac{\text{output}}{\text{input}} \). The selectivity of a join operator is calculated different. The output could always be the Cartesian Product of the two inputs. The selectivity is therefore \( \frac{\text{output}}{\text{input}_{1} \times \text{input}_{2}} \) what is \( \frac{\text{input}_{1}}{\text{input}_{1} \times \text{input}_{2}} \). Applying selectivity routing the Eddy monitors the number of input and return tuples for each operator. These values are used to estimate the operators’ selectivities. The operators are ordered ascending on their selectivity. Tuples are sent to the operator with lowest selectivity first. This operator will drop many tuples which will not have to be processed by any other operator anymore.

The advantage of this strategy is that it reduces the sum of processed tuples over all operators. The disadvantage is that a single operator can become a bottleneck. Imagine two filter operators \( F_{p1} \) and \( F_{p2} \) which are both defined on \( R \)-tuples. Let \( F_{p1} \) have a lower

---

\(^9\)Bit \#0 is assigned to \( F_R \) and bit \#1 and \#2 to the left and right input of \( J_{RS} \).
selectivity than $F_{p_2}$. All tuples are sent to $F_{p_1}$ first. But $F_{p_1}$ might have a poor performance and is not able to process the input immediately. New arriving tuples are buffered until $F_{p_1}$ catches up. At the same time, $F_{p_2}$ is idle because it is performing very fast. We assume that each operator is running in an own thread and that the underlaying hardware has many cores for real parallel execution. This assumption holds for Stratosphere as the system is running in a compute cloud. As the operators are running in parallel, it would be a good idea to send a specific portion of the tuples to $F_{p_2}$ first. This alternative routing order reduces the number of processed tuples $F_{p_1}$ and speeds up overall performance.

In a single-threaded implementation there will be no performance advantage applying this strategy. If the implementation is multi-threaded but there is just one processor, it is not obvious whether concurrent thread execution speeds up performance or not.

### 2.4.3 Back-pressure routing

The back-pressure strategy tries to avoid bottlenecks during query execution in order to keep all operators busy all the time. The size of the input buffers of all operators is monitored. If the size of a buffer grows, the Eddy reduces the number of tuples it sends to it—if the size shrinks, the Eddy routes more tuples to this operator.

Avoiding bottlenecks seems to be a good idea to increase overall performance because tuples are processed and outputted all the time. Hence, overall query processing time is minimized. The disadvantage is that we might do more overall work. A join operator for example might produce many result tuples increasing the number of tuples to be processed by other operators. Recall the example in subsection 2.4.1. In this example there are an $R$-$S$-join and $R$ filter operator in the system. Let us assume a single $r$-tuple which has to be joined with 1,000 $s$-tuples. If the Eddy sends $r$ to the join in the first place, $F$ has to process 1,000 $\langle r, s_i \rangle$ tuples. In this case, an overhead of 999 filter-operations occurs. This filter-operations could be avoided by sending $r$ to $F$ first. If $r$ does not pass $F$, the work of joining $r$ with all $s_i$ is omitted as well.

### 2.4.4 Conclusion

Both discussed routing heuristics from subsections 2.4.2 and 2.4.3 are complementary and both are implemented in [AH00]. The back-pressure strategy is implemented natively by limiting the maximum size of each input buffer for the operators. If an input buffer is full, the Eddy is not able to send the tuple to the operator and has to choose a different one. Selectivity routing is implemented in [AH00] via *Lottery Scheduling* [WW94]. The basic idea is to send a tuple to an operator with a certain probability. This probability is adopted according the operators selectivity as follows: the Eddy credits a ticket to an operator if it sends a tuple to it. If an operator returns a tuple to the Eddy, it gets a ticket deducted. Hence, operators which return little tuples (i.e. have low selectivity) will increase their ticket count over time. When the Eddy has to make a routing decision, it looks at the number of tickets each applicable operator holds. The Eddy holds a lottery and selects an operator with a probability according to the operator’s ticket count. Operators with a low selectivity are chosen with a high probability. It can happen that a tuple will not be

---

10With size we refer to the number of tuples in a buffer and not to the maximal number of tuples a buffer can hold.
send to the operator with the lowest selectivity. But this strategy keeps the Eddy more adaptive because a certain amount of tuples is not routed via the (currently) best order. Therefore, the monitoring behavior of the Eddy is more accurate. If all tuples follow the current routing order, the second and third operators (etc.) do not ‘see’ all tuples but just the remaining tuples from the first operator. Hence, the monitored selectivity of these operators is influenced especially if there is a correlation within the data.

Related Work

Many other papers got published using or improving the behavior of the Eddy. Amol Deshpande [Des04] analyzed the Eddy’s routing overhead. As the Eddy is recalculating the routing order for every tuple, this overhead is quite big. Deshpande showed that the routing overhead can be amortized if re-optimization is triggered at most every tenth tuple. Recalculating the routing order not on a per-tuple basis decreases adaptivity. But as shown in Deshpade’s work adaptivity is sufficient even if re-ordering is triggered for 100 tuples once.

Tian and DeWitt placed the Eddy in a distributed environment [TD03]. The idea is to replace the central Eddy and putting the routing logic into the query operators directly. The operators are linking together in an appropriate way such that they can send tuples to each other directly. Additionally, other routing policies got developed which consider operator selectivity, execution cost, and operator load all together.

Raman, Deshpande, and Hellerstein introduce State Modules as query operators for an Eddy [RDH03]. A State Module (SteM) is basically one half of a join operator. The Symmetric-Hash-Join for example can be represented by two SteMs each hashing one input and probing its tuples against the other SteM. SteMs can also be used to compute \(n\)-ary joins\(^{11}\) (one SteM for each input). Additionally, SteMs expose their internal state to the Eddy which enables the Eddy to make better routing decisions.

Deshpande and Hellerstein developed the STAIR operator for Eddies in [DH04] to improve routing if operators buffer tuples internally. Buffering tuples internally is typical for join operators. If one input relation of a join has an initial delay, the join cannot output any result tuples. The Eddy is usually not aware that one input is delayed. But it monitors the join and gives it a big credit (i.e. many lottery tickets) for its supposed low selectivity and operator costs based on the lottery scheduling algorithm. As the join just processes one input, it performs very fast as it cannot compare tuples. Additionally, it does not return result tuples and hence, the join gets no lottery tickets deducted. Therefore, the Eddy sends even more tuples to the join. When the second input becomes available, the selectivity and operator costs of the join could increase rapidly. But the Eddy is not able to undo its past routing decisions. The wrong routing decisions can slow down query execution massively. The STAIR operator enables the Eddy to undo previous routing steps. This undo operations can improve the average computation time if some inputs have an initial delay.

Trained Eddies (Teddies) are the idea of Claypool and Claypool [CC08]. Teddies reduce the memory overhead by grouping tuples with the same history together. Those tuples

\(^{11}\)A \(n\)-ary join is a join with \(n\) input relations. Most commercial relational database management systems calculate an \(n\)-ary join with \(n - 1\) chained joins which have 2 inputs each.
will have the same state (done- and ready-bit-array for example) which must be stored only once per group. Grouping has the implicit effect that the re-optimization frequency gets reduced as these tuples get routed in one batch. This effect is similar to Deshpande’s study about the Eddy’s routing overhead [Des04].

2.5 The Competition Model

The Eddy approach can be used to alter the order of different query operators. But setting up the Eddy requires to know what operators to use within the system. In the case of Stratosphere, we face the problem not to know what join algorithm is the best for a given PACT program. Hence, we want to use a second adaptive optimization approach to pick the right join algorithm. We extend Antoshenkov’s competition model [Ant93a] for joins. Before we can introduce competing join execution, we explain the details of Antoshenkov’s approach.

Antoshenkov suggested the competition model as an adaptive optimization technique for base table access [Ant93a, Ant93b, AZ96]. Static query optimization faces different problems like cost function instability and sensitivity to variables. Cost estimation errors grow rapidly with the number of join operators [IC91]. But even if the query is fairly simple, query parameters can lead to sub-optimal query execution plans (QEP). Estimating the selectivity of a parameter is very difficult to accomplish because the selectivity can differ rapidly.

Antoshenkov studied selectivity distributions for single and complex predicates for base table access. Many distributions ended up to be crescent-, triangle- or L-shaped—truncated hyperbolas are also very common. Additionally, the skewness increases with decreasing predicate correlation. This distributions mirror a high uncertainty for selectivity estimations of static optimization. The idea of the competition model is to run multiple query plans in parallel for a certain amount of time. During this time the plans are monitored in order to pick the best one. The initial competition time is quite small such that the benefit of picking the better plan is bigger than the overhead of executing multiple plans. We illustrate the idea of the competition model with an example (figure 2.11).

Let $R$ be an input relation with 1,000 tuples and $F_R$ be a filter predicate on $R$ having a query variable. $F_R$ returns 10 tuples or 990 tuples with a probability of 50% each (i.e. there are two different values for the variable and both values are used equally often). We additionally assume that there is an index defined on $R$ which can be used to evaluate $F_R$. Hence, the optimizer can choose between two strategies. Either the complete relation is read (table scan) to find all matching tuples or the index is used (index access). An index returns result tuples only but needs more time for returning a single tuple. The index access performs better if the result has 10 tuples and worse if the result has 990 tuples. For the table scan it is the other way round. If we assume that obtaining a single tuple via index access is five times more expensive as for a table scan, we can compute the average overhead as follows: if the result is 10 tuples the index access produces costs of $10 \cdot 5 = 50$ and the table scan costs $1,000 \cdot 1 = 1,000$. If the result is 990 tuples, the index access costs $990 \cdot 5 = 4,950$ whereas the table scan’s costs are still $1,000$. The average cost for the index scan is therefore $\frac{50 + 4,950}{2} = 2,500$ (2.11(b)) and the average table scan
2.5. THE COMPETITION MODEL

\[
\text{table scan}
\]
(a) \[\text{result is 10 tuples: costs is } 1,000 \times 1 = 1,000\]
(b) \[\text{result is 990 tuples: costs is } 990 \times 1 = 990\]
(c) \[\text{result is 10 tuples: costs is } 10 \times 5 = 50\]

\[
\text{index access}
\]
(a) \[\text{result is 10 tuples: costs is } 10 \times 5 = 50\]
(b) \[\text{result is 990 tuples: costs is } 990 \times 5 = 4,950\]
(c) \[\text{result is 990 tuples: costs is } 1,000 \times 1 = 1,100\]

\[
\text{competing execution}
\]
(a) \[\text{result is 10 tuples: costs is } 10 \times 5 + 50 \times 1 = 100\]
(b) \[\text{result is 990 tuples: costs is } 20 \times 5 + 1,000 \times 1 = 1,100\]
(c) \[\text{result is 990 tuples: costs is } 20 \times 5 + 1,000 \times 1 = 1,100\]

Figure 2.11: Example of a table scan vs. an index access.

Costs \(\frac{1,000+1,000}{2} = 1,000\) (2.11(a)). A static cost based optimizer would choose the table scan strategy due to the lower average costs.

Antoshenkov suggested running both access methods in parallel for a short time. The worse plan is stopped after obtaining 100 tuples in our example. If the result is 10 tuples we get costs of \(10 \times 5 + 50 \times 1 = 100\). After receiving 10 tuples from the index access query execution is finished. Hence, we can shut down the table scan. We assume that we can read five times more tuples with the table scan as with the index access in the same time reflecting the higher costs of using the index. Therefore, the table scan will have returned 50 tuples before the index access finishes. If the result is 990 tuples we get costs of \(20 \times 5 + 1,000 \times 1 = 1,100\). We shut down the index access after receiving 100 tuples from the table scan. The index access has returned 20 tuples in this time. The table scan wins the competition because it performs better in this scenario. Hence, the table scan proceeds and reads the remaining 900 tuples. The average cost in the competition model is \(\frac{100+1,100}{2} = 600\) (2.11(c)), what is 40\% better than the average cost of the table scan (what a static optimizer would choose). If we knew beforehand what the better plan is, we get an average cost of \(\frac{50+1,000}{2} = 525\).

Cost-based static optimization as well as the competition model are heuristic algorithms. The approximation factor of a heuristic is the fraction of calculated result to best possible result. As smaller the approximation factor as better the heuristic works. The static optimizer’s approximation factor is \(\frac{1,000}{525} = 1.90\). The competition model on the other hand approximates with a factor of \(\frac{600}{525}\). This factor is a quite good result as it is very close to the best possible approximation factor of 1.
Chapter 3

The SCORE operator

In the previous chapter we discussed a lot of issues regarding query processing and optimization (including AQP) in a highly parallel and distributed data-flow environment. The aim of Stratosphere is to apply adaptive optimization techniques for PACT program execution. One aspect is re-writing a PACT program or even executing a PACT program using an Eddy operator. The second idea is to use a competition model to find an efficient join algorithms for the three multiple-input PACTs at runtime.

The integration of an Eddy operator within Stratosphere is beyond the scope of this work. But we want to enhance the Eddy by integrating a competition model for joins in it. We call this enhanced Eddy operator SCORE (Stratosphere’s Coordinating Operator for Re-arrangements). Before we can explain the changes needed for competing join execution within an Eddy, we discuss what join algorithms we can run competitive.

3.1 Choosing joins for competitive execution

In sections 2.2 and 2.3 we discussed different join algorithms. It is not straight forward what join algorithms should be executed competitive. From a theoretical point of view, algorithms which have about the same time complexity should compete. The NLJ for example has quadratic time complexity whereas the HJ needs linear time (in most practical cases). Hence, the HJ is expected to win always and a competitive execution is mandatory. The HJ and SMJ do not have the same time complexity. But the hashing and probing overhead (including key-collisions) of the HJ can be severe. Depending on the data distribution of the input the SMJ might perform better than the HJ.

We divided the existing join algorithms into two main groups: blocking and symmetric (pipeline) join algorithms. Due to the initial delay of blocking joins a competing execution of a blocking and a symmetric join algorithm is not applicable. Hence, we identify the HJ and the SMJ for competing execution if the result is small (i.e. has linear size). If the result is expected to be the Cartesian Product (or has at least quadratic size) an complementary execution of the NLJ and the HJ, the NLJ and the SMJ or the SNLJ and the SHJ are applicable as well.

We introduced five join algorithms in sections 2.2 and 2.3 (NLJ, HJ, SMJ, SNLJ, SHJ). The NLJ is the only algorithm which can evaluate any join predicate. Hence, the NLJ is
important as Stratosphere needs to deal with unknown user-defined \( JOIN(.,.) \) functions. A \( JOIN(.,.) \)-call might be very cost intensive. Hence, we developed the new Bucket-Nested-Loop-Join (BNLJ) algorithm. It is also able to deal with any join predicate and might perform better than the NLJ. Both algorithms have the quadratic time complexity. Hence, we can run them competitive. How the BNLJ works in details follows in the next subsection.

### 3.1.1 Bucket-Nested-Loop-Join

In this section we introduce a new join algorithm. We call it Bucket-Nested-Loop-Join (BNLJ). It is a novel approach to improve the performance of a NLJ if an unknown (user-defined) \( JOIN(.,.) \) function is provided. The basic idea is to reduce the number of \( JOIN(.,.) \) calls by hashing both inputs and perform the nested-loop over all buckets. The \( JOIN(.,.) \) function is called for every bucket pair once. If it returns \( true \), all tuples from both buckets are joined together.

While the NLJ has a complexity of \(|R| \cdot |S|\), the BNLJ reduces the number of comparisons to \(|h_R| \cdot |h_S|\), where \( |h_R| \) and \( |h_S| \) are the number of buckets of hash table \( h_R \) and \( h_S \) respectively. Figure 3.1 shows a NLJ on the left hand side. Each arrow represents a \( JOIN(.,.) \) call. The right hand side shows the BNLJ. Both hash tables are build up already and \( JOIN(.,.) \) is called for each bucket pair once. The number of buckets

---

1The function \( JOIN(.,.) \) takes two tuples as input (both of each relation) and returns \( true \) if the two input tuples satisfy the join predicate—\( false \) otherwise (as defined in subsection 2.2).
3.1. CHOOSING JOINS FOR COMPETITIVE EXECUTION

for each tuple \( r \) in \( R \) do
\[ h_R(r) := h_R(r) \cup \{r\} \]
endfor

for each tuple \( s \) in \( S \) do
\[ h_S(s) := h_S(s) \cup \{s\} \]
endfor

for each bucket \( b_R \) in \( h_R \) do
for each bucket \( b_S \) in \( h_S \) do
\[ \text{if} \ JOIN(\text{choose}(b_R), \text{choose}(b_S)) = \text{true} \text{ then} \]
\[ \text{for each tuple} \ r \text{ in } b_R \text{ do} \]
\[ \text{for each tuple} \ s \text{ in } b_S \text{ do} \]
\[ \text{output} \ \langle r, s \rangle \]
\[ \text{endfor} \]
\[ \text{endfor} \]
\[ \text{endif} \]
endfor
endfor

Figure 3.2: Bucket-Nested-Loop-Join-Algorithm.

is expected to be smaller than the number of tuples. Hence, \(|h_R| \cdot |h_S| < |R| \cdot |S|\). If this assumption does not hold, the BNLJ is expected to perform worse than the NLJ because hashing both inputs costs some overhead. On the other hand, if the assumption holds, the reduced number of \( JOIN(\ldots) \) calls compensates this overhead and results in a performance advantage compared to the NLJ.

The BNLJ is expected to perform better compared to the NLJ as the user-defined \( JOIN(\ldots) \) function is more difficult. If the function is more difficult, it consumes more time and hence, the BNLJ saves more time. Additionally, the BNLJ performs better as the number of buckets is smaller, reducing the number of comparisons.

Compared to the NLJ the BNLJ has a higher memory consumption. As the BNLJ hashes both inputs instead of buffering just the inner relation, \( C_S \) is \( O(|R| + |S|) \) instead of \( O(|S|) \). The pseudo-code example in figure 3.2 illustrates the BNLJ in more detail. First, both relations get hashed completely into separate hash tables (lines 1–4 and 5–8). Afterwards a nested-loop is performed over all buckets in which the join result is calculated.

3.1.2 Symmetric-Bucket-Nested-Loop-Join

The pseudo-code in figure 3.2 shows the blocking version of the BNLJ. We discussed the pros and cons of blocking join algorithms in section 2.3. To avoid the disadvantages of a blocking algorithm we developed a symmetric version of the BNLJ as well. The most difficult part of it is the book keeping overhead. The idea of the BNLJ is to reduce the number of comparisons. This reduction is easy to accomplish if both relations get hashed first. In the symmetric version hashing and probing are interleaving. Hence, we have to
function getBucket(t,h₁,h₂,flip)
  b := h₁(t)
  if b.S = {} then /* indicates that b is a new bucket */
    for each bucket b' in h₂ do
      if flip = false then
        if JOIN(b,k,b'.k) = true then
          b.L = b.L ∪ b'
          b'.L = b'.L ∪ b
        endif
      else
        if JOIN(b'.k,b.k) = true then
          b.L = b.L ∪ b'
          b'.L = b'.L ∪ b
        endif
      endif
    endfor
  endif
  b.S = b.S ∪ {t}
  return with b
end function

r := next(R)
s := next(S)

while r ≠ EOS or s ≠ EOS do
  if r = null then
    r := next(R)
  endif
  if r ≠ null and r ≠ EOS then
    b := getBucket(r,h_R,h_S,false)
    for each bucket b' in b.L do
      for each tuple t in b'.S do
        output ⟨r,t⟩
      endfor
    endfor
    r := next(R)
  endif
  if s = null then
    s := next(S)
  endif
  if s ≠ null and s ≠ EOS then
    b := getBucket(s,h_S,h_R,true)
    for each bucket b' in b.L do
      for each tuple t in b'.S do
        output ⟨t,s⟩
      endfor
    endfor
    s := next(S)
  endif
endwhile
3.2. COMPETING JOINS

keep track what bucket-pairs are matching because we do not want to check a bucket-pair twice. A bucket-list is added to each bucket. This list contains all joining buckets of the other hash table. If a new tuples is added to a bucket, we step though the list of buckets and join the new tuple with all tuples from all buckets in the list.

If we add the first tuple to a bucket, we have to do a loop over all buckets from the other hash table to build up this list. Building up the bucket-list includes adding the new bucket to the lists of all joining buckets, i.e. we link in both directions. The pseudo-code example (figure 3.3) of the Symmetric-Bucket-Nested-Loop-Join (SBNLJ) uses the following bucket structure: \( \text{bucket} := (k, S, L) \). \( k \) is the key of the bucket and \( S \) is the set of tuples the bucket contains. \( L \) is the list of joining buckets of the other hash table (i.e., for a bucket \( \langle k, S, L \rangle \) in \( h_R \): \( L := \{ \text{bucket} \in h_S | \text{JOIN}(k, \text{bucket.k}) = \text{true} \} \)). The function \( \text{getBucket}(t, h_1, h_2, \text{flip}) \) returns the bucket to which tuple \( t \) belongs to. If \( t \) is the first tuple of the bucket (i.e. we get a new bucket), the list of joining buckets of the other hash table is build up within \( \text{getBucket()} \) as a side-effect. The parameter \( \text{flip} \) indicates to which hash table \( t \) belongs (\( h_r \) or \( h_s \)). This is important as we cannot assume that the function \( \text{JOIN}(\ldots) \) is commutative.

3.1.3 Summary

In this section we discussed join algorithms we want to run competitive and introduced two new join algorithms for competing execution (BNLJ and SBNLJ). If the join result is expected to be big (i.e. has quadratic size), we can run the (Symmetric-)Nested-Loop-Join against the (Symmetric-)Hash-Join or the Sort-Merge-Join. Additionally, the competing execution of the Hash-Join and the Sort-Merge-Join is applicable. If an unknown user-defined \( \text{JOIN}(\ldots) \) function specifies the join condition, we can apply our competition model to the Nested-Loop-Join and the Bucket-Nested-Loop-Join.\(^2\) The details how our join competition model works follows in the next section.

3.2 Competing Joins

We explained Antoshenkov’s competition model in section 2.5. In this section we extend it for running joins competitive and integrate the join competition model into the Eddy operator. The main difference of running joins competitive to Antoshenkov’s approach is that join algorithms have an internal state. Processed tuples are buffered internally and the algorithm’s state is growing over time. The performance of a join algorithm might change over time due to its growing internal state. A good example is the Symmetric-Nested-Loop-Join (subsection 2.3.2). For each arriving tuple its loops over all tuples of the internal tuple buffer of the other input. As more tuples are buffered internally as longer it takes to finish this loop. Hence, the processing time increases for each new tuple. In Antoshenkov’s model, on the other hand, the single access methods are assumed to have a stable performance what make the model simpler.

3.2.1 Competing join execution

To be able to compare the runtime behavior of two algorithms exactly it is necessary to execute both with the same data. As both algorithms have to run independent from each

\(^2\)We can also use the Symmetric-Nested-Loop-Join and the Symmetric-Bucket-Nested-Loop-Join.
other, it is necessary to duplicate the input such that each algorithm works on its own data set. Duplicating the input has two disadvantages. First, it is quite cost intensive to replicate the input. Second, both operators are calculating the same logical join. That implies that the result is calculated twice what is a waste of resources. Additionally, duplicates violate the semantics of a join operation and must be deleted therefore. Duplicate deletion adds an additional overhead to query processing. The easiest way to accomplish duplicate elimination is to block the output until the first operator finished. The second operator is terminated and its computed output gets dropped. The output of the first operator is returned as result finally.

Blocking the result until the first operator finishes is suboptimal for our data-flow environment as it may result in delays which influence the overall query execution time (see section 2.3). If we do not want to block the result, we need to keep track what tuples got already returned. If a tuple is going to be returned a second time, the action has to be suppressed. One way to accomplish suppression is to keep a tuple identifier (TID) for each returned result tuple and compare the TID of all consecutive tuples with this list of TIDs. If the TID of a returned tuple is already in the list, it can be deleted. Otherwise the tuple is returned and its TID is added to the list. The problem is that this strategy
3.2. COMPETING JOINS

implies a big computation and memory overhead. That is the reason why we prefer a strategy which avoids duplicates from the beginning.

Instead of duplicating the input we divide it equally over both operators (figure 3.4(a)). Let \( R \) and \( S \) be the two input relations. We divide \( R \) into \( R_1 \) and \( R_2 \) with \( R = R_1 \cup R_2 \), \( R_1 \cap R_2 = \emptyset \) and \( |R_1| \approx |R_2| \). \( S \) is divided into \( S_1 \) and \( S_2 \) similarly. Now we send \( R_1 \) and \( S_1 \) to the first operator \( Op_1 \) and \( R_2 \) and \( S_2 \) to the second operator \( Op_2 \). \( Op_1 \) returns different result tuples than \( Op_2 \) because it works on different data. Hence, we do not need to delete duplicates. But we do not get the complete result so far. The join between \( R \) and \( S \) can be expressed in relational algebra as follows:

\[
R \bowtie S = \left( R_1 \cup R_2 \right) \bowtie \left( S_1 \cup S_2 \right) = R_1 \bowtie S_1 \cup R_1 \bowtie S_2 \cup R_2 \bowtie S_1 \cup R_2 \bowtie S_2
\]

If we divide the input as described, we calculate \( R_1 \bowtie S_1 \cup R_2 \bowtie S_2 \) but miss \( R_1 \bowtie S_2 \) and \( R_2 \bowtie S_1 \). We solve this problem as follows: let \( Op_1 \) be the operator which finishes first. Hence, we are going to shut down \( Op_2 \). As \( Op_1 \) is the better performing operator we use it to calculate the missing result. First, we move all unprocessed tuple from \( R_2 \) and \( S_2 \) to \( Op_1 \) as regular input (grey in figure 3.4(b)). These tuples have not been processed by \( Op_2 \) and can be joined regularly by \( Op_1 \). Afterwards the processed tuples \( R_2 \) and \( S_2 \) are moved from \( Op_2 \) to \( Op_1 \). \( Op_1 \) calculates \( R_2 \bowtie S_1 \) and \( R_1 \bowtie S_2 \) than. \( Op_1 \) must not calculate \( R_2 \bowtie S_2 \) because is was calculated by \( Op_2 \) already.

3.2.2 Competition within SCORE

SCORE integrates the competition strategy into an Eddy. Hence, SCORE’s internal behavior must be adopted. The first thing is to make SCORE aware of competing operators. We introduce Operator Groups (OpGrp). An OpGrp contains exactly two operators which
are either running competitive or complementary. Figure 3.5(a) shows the logical setup of an OpGrp linked to SCORE. Even if the OpGrp is logically calculating one join, we need to link both internal join operators to SCORE as shown in 3.5(b). SCORE has two different routing strategies for the two different OpGrps. If an OpGrp is running competitive, SCORE partitions the input over both operators of the group as described in our join competition model in subsection 3.2.1. It sends all even tuples (1st, 3rd, 5th ...) to the first and all odd tuples (2nd, 4th, 6th ...) to the second operator. This strategy assures that both operators get the same number of tuples as input. The basic strategy of the join competition model is to monitor both operators until one of them has processed all tuples. When the first operator finishes, we know what operator has the better performance. SCORE shuts down the worse operator and moves all internally buffered tuples from it to the better performing one. This group shut-down happens concurrently (or even parallel) to the regular routing process, i.e. the system does not wait until the tuples got moved. The only restriction which applies is that SCORE cannot route new tuples to the OpGrp while one operator is shut down and the tuples are moved. While moving the tuples, the OpGrp returns the newly joined tuples immediately to SCORE. Hence, from SCORE’s point of view the operator is just too busy to receive new tuples. After the OpGrp got shut down, it is replaced by the winning join operator (figure 3.6).

3.3 Complementary Joins

Additionally to competing joins we introduce a complementary join algorithm. Our complementary join algorithm consists of two sub-joins each computing a dedicated part of the join result. Complementary join algorithms work different to competing joins where both algorithms calculate the same result. In order to perform a complementary join we

\footnote{The complementary Merg-Hash-Join algorithm and more details about complementary OpGrps follow in subsection 3.3.}
need to alter the tuple distribution strategy. We do not divide the input equally over both operators like in the competing case. Additionally, it is naturally not necessary to handle duplicates explicitly because the algorithms calculate a distinct result set anyway. The complementary distribution strategy (which will be explained in the following subsection) divides the input in a way such that each algorithm calculates a dedicated part of the result.

### 3.3.1 Merge-Hash-Join

In this section we introduce the *Merge-Hash-Join* (MHJ) algorithm. It is a complementary algorithms consisting of a Merge-Join (MJ) and Symmetric-Hash-Join (SHJ) part. We want to use the MJ in our system as it has the best time complexity over all introduces join algorithms. The MJ requires a sorted input which in not available often. In data-flow environments like Stratosphere there are cases in which the input is almost sorted. We want to exploit the order by applying a MJ to almost sorted data. We alter the Merge-Join into an *Optimistic-Merge-Join* (OMJ) which is explained in detail in the next paragraph. All tuples which cannot be processed by the OMJ (because they are not in sorting order) are processed by a modified SHJ which runs complementary to the OMJ. We call this modified SHJ *Auxiliary-Symmetric-Hash-Join* (ASHJ). The Merge-Hash-Join is linked to SCORE as an complementary Operator Group. How SCORE handles complementary OpGrps is explained in subsection 3.3.2. In the next two paragraphs we explain the details about the OMJ and ASHJ.

**Optimistic-Merge-Join**

The *Optimistic-Merge-Join* (OMJ) is a modified MJ. It has a filter component at both inputs. The filter components check if the tuples are in-order. If a tuple inside the stream is not in-order the tuple gets rejected. Hence, the input for the merge part is sorted correctly as tuples which violate the sorting order are rejected, i.e. sent back to the SCORE. We call the tuples which are rejected by the OMJ *out-of-order-tuples* (ooo-tuples). All other tuples are called *regular* or *in-order*-tuples. Figure 3.7 shows the conceptual setup of the OMJ part of the complementary Merge-Hash-Join linked to SCORE. A detailed explanation how the Auxiliary-Symmetric-Hash-Join (ASHJ) and is linked to SCORE follows. The OMJ is linked to SCORE twice. SCORE has to distinguish between the left and right input. Hence, both inputs are linked separately what implies that two done- and ready-bits are needed. The shown return paths (the two rejects and the output) are actually just a logical distinction. All rejected and returned tuples are added to SCORE’s internal tuple buffer.

The filters for the ooo-tuples of both inputs are applying the following basic heuristic: if the current tuple is less than the last processed tuple the tuple is rejected. This algorithm is straight forward but has a shortcoming which is illustrated with an example. Let the input be 1, 2, 3, 5, 4, 6, 10, 7, 8, 9. The heuristic rejects 4 because it is less than its predecessor 5. The filter also rejects 7, 8, 9 because they are less than 10. But actually we want to reject as little tuples as possible. The MJ is expected to be the better performing algorithm and should process as many tuples as possible. Hence, it would be a better choice to reject 10 but process 7, 8, 9 (which are greater than 6). We must alter the filter strategy such
that the filter is able to reject some tuples which are greater than the last processed one. We enhance the heuristic as follows: we introduce a look-ahead-buffer of three tuples to compare the current tuple with its successors. The buffer works with the first-in-first-out (FIFO) logic. When a new tuple are added to the buffer, we check if it is greater than the last processed tuple. If not, the tuple is rejected. Additionally, we ensure at all times that the tuples in the look-ahead-buffer are greater than the last processed tuple (we call it the greater-invariant). If we are going to process the next tuple, we check if it is greater than all three look-ahead-tuples.\footnote{If the look-ahead-buffer has not three tuples in it and we can not fill it up because the input delays, we still keep processing and compare the tuple with the remaining look-ahead-tuples.} If it is greater, the tuple gets rejected because we assume that we can process the remaining three tuples instead of this single tuple.

If the tuple is not rejected but returned for processing, it can happen that the look-ahead-buffer violates the greater-invariant afterwards. Figure 3.8 shows an input stream with a look-ahead-buffer of three tuples. Imagine the last tuple had value 3 and the tuples in the look-ahead-buffer are 5, 4, 6 (figure 3.8(a)). All tuples got added to the buffer as they are greater than the last processed tuple (which was 3). Now we get the next tuple from the buffer (figure 3.8(b)). Hence, we are talking 5 out of the buffer and add the next tuple from the input to the buffer. The next tuple is 10 and can be added to the buffer as it is greater than 5. The look-ahead-buffer has now 6, 4, 10 in it. 5 is not greater than all three tuples in the buffer and is processed regularly by the MJ. The last processed tuple is now 5. Hence, the greater-invariant of the look-ahead-buffer is violated now because 4 is less than 5. We have to remove 4 from the buffer and reject it to maintain the greater-invariant (figure 3.8(b)).\footnote{This operation violates the FIFO logic but is necessary to ensure correctness.} Afterwards the buffer is filled up with the next tuple to keep a

\begin{figure}
\centering
\includegraphics[width=\textwidth]{fig3_7.png}
\caption{The conceptual setup of an MHJ linked to SCORE.}
\end{figure}
3.3. COMPLEMENTARY JOINS

look-ahead of three tuples. Now we process tuple 6 regularly (figure 3.8(c)) as all tuples in the cleaned-up look-ahead-buffer are greater than it. The next tuple to process is 10 (figure 3.8(d)). But as 10 is greater than all three look-ahead-tuples, it is rejected. Tuples 7, 8 and 9 are finally processed by the MJ. We see that this advanced heuristic with a look-ahead-buffer of three tuples just rejects two tuples (4 and 10) instead of four tuples (4, 7, 8 and 9) like the basic filter strategy. We choose a look-ahead of three tuples because a smaller or bigger look-ahead seems not to work appropriate. If the look-ahead is smaller, it is more likely that a tuple is bigger than all look-ahead tuples. Hence, more tuples would get rejected but we want to reject as little as possible. If the look-ahead is bigger, the reject condition is stronger. In our example the next tuple might be 11 most likely and 10 would not get rejected applying a bigger look-ahead. This can lead to even more rejected tuples in following filter steps. In our example tuples 7, 8, 9 would be rejected if the next tuple is 11 and we apply a look-ahead of four tuples.

**Auxiliary-Symmetric-Hash-Join**

The OMJ rejects tuples which are out-of-order. Hence, it can happen that the OMJ misses result tuples. To complete the result we use a modified Symmetric-Hash-Join which calculates the missing part of the join. If an ooo-tuple is in the stream, matching join tuples could be located at any other position in the other stream. Hence, the SHJ needs the complete data as input to be able to calculate the missing result tuples. As both sub-joins of the Merge-Hash-Join (OMJ and modified SHJ) needs the complete data as input, we have to duplicate it such that each algorithm works on its own data set. Additionally, we have to alter the SHJ. Otherwise it would calculate the complete result as it gets the complete data as input. We call the modified version Auxiliary-Symmetric-Hash-Join (ASHJ). Instead of hashing and probing each tuple the ASHJ builds up two

---

6 We want to process as many tuples as possible by the MJ because of its better time complexity.
different hash tables for both inputs and does just some specific probe operations which are described in the next paragraph. The two hash tables per input are a main hash table where all regular (in-order-tuples) get hashed. The second hash table is a hash table for ooo-tuples. The two hash tables partition the input into two subsets each. The four hash tables divide the join result into four sub-results. Let \( R \) and \( S \) be the two input relations. \( R_M \) and \( S_M \) are the two main subsets (the tuples which are in-order). \( R_{oo} \) and \( S_{oo} \) are the subsets containing the ooo-tuples. The join result can now be expressed as \( R \bowtie S = R_M \bowtie S_M \cup R_M \bowtie S_{oo} \cup R_{oo} \bowtie S_M \cup R_{oo} \bowtie S_{oo} \). The main part \((R_M \bowtie S_M)\) is calculated by the OMJ. Hence the ASHJ has to calculate the three remaining parts.

The rejected tuples from OMJ get marked as rejected by the OMJ and are sent back to SCORE. SCORE routes these tuples to the ASHJ afterwards (additionally to the regular input of the ASHJ). Hence, ooo-tuples are sent to the ASHJ twice. The first time the ASHJ does not know that the tuple is an ooo-tuple because it is not marked (SCORE just replicated the tuple and sent it to both join algorithms). The ASHJ inserts this tuple into the main hash table first. If the ASHJ processed the corresponding ooo-tuple (the tuple rejected by OMJ and sent to ASHJ by SCORE a second time), it corrects this mistake by removing the tuple from the main hash table and putting it into the ooo-hash-table where the tuple belongs to. Figure 3.9 shows an example of the ASHJ. The input for both relations \( R \) and \( S \) is the same as in the ooo-filter-example from above (1, 2, 3, 5, 4, 6, 10, 7, 8, 9). We see that the ASHJ has already hashed some tuples (1, 2, 3, 5, 4 for \( R \) and 1, 2, 3, 5 for \( S \)) into the main hash tables \((h_M)\) as none of these tuples was marked as rejected (figure 3.9(a)). The ooo-hash-tables \((h_{oo})\) are both still empty. Tuple 4 got rejected (and marked rejected—subscript \( r \) in our example) by the filter of the OMJ. SCORE sends tuple 4\(_r\) to the ASHJ right after sending tuple 4 to it in our example. Let the ASHJ process 4\(_r\) from input \( R \) now (figure 3.9(b)). As 4\(_r\) is marked as rejected, 4 gets moved from \( h_M \) to \( h_{oo} \). Afterwards 4 is probed against \( h_M \) from input \( S \) (where no matching tuple is found). The rejected tuple 4\(_r\) is a duplicate from 4 and is dropped by the ASHJ. In the next step ASHJ processes tuple 4 from input \( S \). 4 is not marked as rejected and is inserted into \( h_M \) (figure 3.9(b)). Than 4 is probed against \( h_{oo} \) of input \( R \). This probe operation implicitly calculates \( R_{oo} \bowtie S_{oo} \) in our example (tuple 4 is actually an ooo-tuple). Hence, ASHJ needs just two different probe operations to calculate the three missing parts of the join result \((R_M \bowtie S_{oo}, R_{oo} \bowtie S_M \text{ and } R_{oo} \bowtie S_{oo})\). A probe operation into the ooo-hash-table is not necessary because this part of the result is calculated implicitly. Afterwards tuple 4\(_r\) from \( S \) is processed (figure 3.9(c)). Hence, 4 is moved from \( h_M \) to \( h_{oo} \) and probed against \( R \)'s \( h_M \).

In general, ASHJ probes each new arriving regular tuple against the ooo-hash-table of the other input after inserting it into the main hash table. Additionally, it probes all ooo-tuples against the main hash table of the other input after removing it from its own main hash table and inserting it into the ooo-hash-table. We see that ooo-tuples get probed against the ooo-hash-table when they get inserted (by mistake) into the main hash table. At this point in time it was not clear that the tuple is out-of-order. When the tuple is processed a second time (now marked as out-of-order), it was already probed against the ooo-hash-table. Hence, we do not have to probe it a second time.
3.3. COMPLEMENTARY JOINS

3.3.2 Complementary joins within SCORE

If an OpGrp is running complementary (in opposite to competing—subsection 3.2.2), SCORE replicates the input and routes all tuples to both operators of the group. The first operator can reject tuples by marking them as rejected and sending them back to SCORE. SCORE knows now that it has to send these tuples to the second operator of the complementary OpGrp. The main difference of complementary OpGrps to competing OpGrps is the following: competing groups are designed such that one operator is going to be shut down while the other operator finishes the work. Complementary groups in opposite are designed such that both operators are running until query execution finishes.

We introduced the complementary Merge-Hash-Join (MHJ) in subsection 3.3.1. The semantic of our MHJ is that the first operator (Optimistic-Merge-Join) can calculate a specific part of the result. The second operator (Auxiliary-Symmetric-Hash-Join) is actually able to calculate the complete result (in the case that the Merge-Join rejects all tuples). That the second operator can calculate the complete result is a special case. This special case enables us to shut down the first operator of a complementary group. We can apply the following shut down strategy: the number of rejected tuples from the Optimistic-Merge-Join is monitored. If the number is bigger than a certain threshold (e.g. 50%), we shut down the OMJ. The Auxiliary-Symmetric-Hash-Join now performs like a regular Symmetric-Hash-Join calculating the complete join result. The overhead of routing tuples to the OMJ which rejects most tuples before they can be sent to the ASHJ

![Diagram of complementary joins within SCORE](image)
is avoided now. From SCORE’s point of view the complementary OpGrp is replaced by a single join operator.

It is not clear if this shut down strategy can be applied to other complementary OpGrps as well. Whether it can be applied or not, depends on the semantics of the OpGrp, i.e. if the two complementary executed join algorithms work together in the same way as the OMJ and the ASHJ. In general, SCORE does not assume that it can shut down one of the two complementary algorithms but run both until query execution finishes.

### 3.4 SCORE’s routing strategies

The routing strategy and the overall behavior of SCORE is designed to support data streams as inputs and pipeline operators only. However, we are using SCORE for query processing like a relational database management system does. Hence, we have to introduce a few new concepts for SCORE. First, we introduce markers within the data streams. The first marker is called *End-of-Stream* (EOS). An input stream sends an EOS marker to SCORE after it has sent all data tuples to it. Sending an EOS marker to SCORE is the last action of an input stream before it terminates. If SCORE receives an EOS from an input, it knows that it received all tuples from it. SCORE keeps this information. After all input streams sent an EOS to SCORE, SCORE knows that no new input tuples will arrive. It waits until all operators sent back all result tuples. After receiving all result tuples and sending them to the output stream, SCORE can terminate the operators (by sending EOS markers to them as well). An operator will terminate if it receives an EOS marker because that indicates that it has processed all tuples. After all operators finished processing, SCORE sends a final EOS marker to the output stream and terminates itself. The EOS indicates that the query result is complete. The output stream terminates and query processing is finished.

#### 3.4.1 Blocking operators within a SCORE system

Additionally to finite inputs we use blocking operators within SCORE (e.g. Sort-Merge-Join). The above described shut down strategy using EOS markers fails if there are blocking operators in the system. Blocking operators do not return any result tuples to SCORE until they received all data (from at least one input). Hence, SCORE waits for the operator until it has processed all data before sending the EOS marker. But the operator does not know that it has received all data already because the EOS marker which indicates that has not been sent to it. We resolve this deadlock situation by making SCORE aware of blocking operators within the system. If all inputs terminated and all operators (except the blocking operator) have returned all result tuples to SCORE, it sends an EOS marker to the blocking operator. We do not violate any semantics because the complete input is buffered within the blocking operator at this point in time. The blocking operator calculates the correct result and returns it to SCORE. As the complete input was buffered within the blocking operator, it will not receive any more tuples from SCORE. Therefore, the blocking operator terminates as well. SCORE routes the new intermediate result tuples to all other operators now. As there are no blocking operators in the system anymore, we can now apply the regular shut down strategy.
3.4. SCORE’S ROUTING STRATEGIES

Sending the EOS marker to the blocking operator as described works only if there is exactly one blocking operator in the system. If there are multiple blocking operators, this strategy fails. Each blocking operator has a part—but not the complete input—in its internal buffers. Hence, sending an EOS marker to them violates the operator’s assumption that it has the complete input in its buffers. The blocking operators will not calculate the complete result. This problem can be solved by limiting the number of blocking operators to a maximum of one.

The limitation to have just one blocking operators in the system implies that we cannot run blocking operators within a competition OpGrp. The input gets divided over both operators of the group. Dividing the input over multiple blocking operators violates the blocking operator assumption to have the complete input in its buffers (this case is similar to a SCORE system with multiple blocking operators). To solve this problem we introduce (data) partitions and alter SCORE as well as the blocking join algorithms to be partition-aware. This new concepts are described in the following section.

3.4.2 Partition-aware blocking join operators

A partition-aware blocking join operator receive the input data not at once but partition wise. Therefore, it executes the regular join algorithm for each partition pair once. Let the join operator have two inputs \( R \) and \( S \) which are partitioned into three parts each. The join is calculated as follows: after receiving the first two partitions \( R_1 \) and \( S_1 \), both partitions are joined regularly. In the following step the next partitions \( R_2 \) and \( S_2 \) are received and joined. However, the algorithm knows that it received \( S_1 \) and \( R_1 \) already (which are still buffered). Hence, it executes the join for the partition pairs \( R_1 \) and \( S_2 \) as well as \( R_2 \) and \( S_1 \), too. The same happens after joining \( R_3 \) with \( S_3 \) by joining \( R_1 \) with \( S_3 \), \( R_2 \) with \( S_3 \) as well as \( R_3 \) with \( S_1 \) and \( R_3 \) with \( S_2 \) (figure 3.10).

The concept of partitions for blocking operators can also be modeled as first- and second-order functions. The actual join-algorithm is the first-order function. The partition-aware part is the second-order function taking data partitions and the join-algorithm as input. The second-order function calls the join-algorithm for every partition pair of the input. Hence, the actual join algorithm does not need to be modified but is just wrapped with the partition-aware part.

To use multiple partition-aware blocking operators within SCORE we introduce a stream marker called *End-of-Partition* (EOP). SCORE can send such a marker to blocking join operators at any point in time. The operator which receives an EOP marker treats the currently buffered tuples as one partition and performs the above described join strategy. We advance SCORE’s shut down strategy by using EOP markers. Instead of sending an EOS tuple to all blocking operators SCORE sends EOP markers to them. EOP markers might be sent to one blocking operator multiple times depending on the number of blocking operators within the SCORE system. A blocking operator process a data partition each time it receives a EOP marker. Query processing is finished if no data tuples got send to blocking operators by the SCORE after EOP markers got sent to them. If the EOP markers got send to the blocking operators in the last, SCORE sends EOS markers to

---

\(^7\)The altered strategy including EOP markers is a generalization and works for a single blocking operators as well.
CHAPTER 3. THE SCORE OPERATOR

Figure 3.10: Example of a partiton-aware blocking join.

all operators in any order without violating any assumptions. The pipeline as well as the blocking operators have already calculated the complete result and can terminate immediately after receiving the EOS marker.

Figure 3.11 shows a SCORE setup with two blocking join operators. The three input streams $R$, $S$ and $T$ are not shown. The first join $J_{RS}$ computes $R \bowtie S$; the second join $J_{ST}$ computes $S \bowtie T$. SCORE routes the complete input of $R$ to $J_{RS}$ and the complete input of $T$ to $J_{ST}$. Relation $S$ is divided equally over both operators. After the routing is finished, SCORE sends EOP markers to the joins which start to process the given partitions (figure 3.11(a)). The intermediate results $RS_1$ and $S_2T$ are returned to SCORE. SCORE routes the tuples as second partitions to the operators. After both intermediate results are calculated and routed completely, SCORE sends EOP markers to the operators again (figure 3.11(b)). The new partitions for input $R$ of $J_{RS}$ and $T$ of $J_{ST}$ are now empty. Hence, $J_{RS}$ needs join $R$ with $S_2T$ only—$J_{ST}$ calculates just $RS_1 \bowtie T$. The two sub-results $RS_2T$ and $RS_1T$ are returned to SCORE as final result. After sending all $RST$-tuples to the output, SCORE sends EOS markers to both operators to terminate them. Finally, it sends an EOS marker to the output-stream and query execution finishes.

EOP markers and the partition-strategy enable the SCORE to deal with multiple blocking operators. The disadvantage is that the complexity of the used join algorithm increases with an increasing number of partitions. In the case of the Hash-Join for example and three partitions as in figure 3.10 each tuple is probed three times instead of once. If a Sort-Merge-Join is used, there is even more work to do. The actual merging has to be done for each partition pair. Hence, we access each tuple $3^2 = 9$ times (the complexity is quadratic in the number of partitions).

3.4.3 Competition of blocking joins

Having the concept of partitions for blocking operators we can use this concept for blocking competing OpGrps. SCORE divides the input symmetrically over both operators of the blocking OpGrp. If SCORE sends an EOP marker, the marker is sent to both operators of the group. We can now apply the basic monitoring strategy to the OpGrp. We wait
until one operator finishes processing and shut down the other one. Afterwards we move the tuples from the worse operator to the better performing operator. The data from both inputs is treated as single partitions within the winning operator. Hence, the better operator joins the new partitions correctly with the partitions it received already.

### 3.4.4 Advanced competition strategy

The describe shut down strategy for competing OpGrps is too straight forward and not adaptive enough. There are two problems related to it: 1) Waiting until the first operator finishes processing implies that we wait until the complete input was processed. Hence, we delay the decision about the better performing operator too long. We want to shut down one operator as early as possible to gain a bigger performance advantage. 2) If we wait until the first operator finished processing, each operator buffers about 50% of the both inputs internally. If we shut down the worse operator now, we need too much time to shift the tuples from the worse to the better operator. Moving the tuples to the better performing operator includes joining them with the operator’s buffered tuples. The winning operators calculated about 25% of the complete result as it received half the data. The loosing operator calculated less than 25% of the result as it could not finish its work. Hence, the tuple moving- and joining-step calculates more than 50% of the result. Therefore, the shifting time can be expected to be about twice the amount of time the first operator needed to finish. We shut down the worse operator after about 33% of query execution time.
Basically, we can terminate a group at any point in time. We can apply the same metric as described for the basic shut down strategy to blocking operators. If we want to terminate a blocking group, we send an EOP marker to the group. We can now wait until one operator terminates and shut down the other one. This strategy can be applied with a more or less aggressive style. The least aggressive style is to wait until the regular shut down strategy gets applied by SCORE anyway. Alternatively we could wait until e.g. 20%, 10% or 5% of the input got buffered within the operator. Additionally, a fixed amount of tuples (e.g. 100,000) could be the threshold before sending the first EOP marker. It would also be possible to terminate a group independent from the amount of already processed data but after a certain amount of time (e.g. a few millisecond or seconds—or maybe a certain percentage of expected query execution time).

In the case of symmetric join algorithm SCORE cannot use EOP markers as symmetric join operators are not able to deal with them. Instead SCORE just stops sending tuples to a OpGrp after the tuple threshold is reached. SCORE waits until one operator finishes processing and shuts down the other one. An alternative is to compare the number of tuples in the input buffers of the operators after the threshold is reached. Symmetric join algorithms process available tuples immediately. If we assume to have unlimited buffers, the number of tuples in the input buffers indicates what operator performs better: the operator with less tuples in the buffers. If the input buffers are not unlimited, comparing the number of tuples in the buffers might not work appropriate (depending on the size of the buffers). The reason is that the number of tuples in a buffer (if the buffer is small) changes very quickly, i.e. it tends to oscillate between empty and full. This oscillation makes buffer size comparison not applicable. But in the case of limited input buffers, the number of buffer overflows can be counted. A buffer overflow occurs if the buffer is full. The operator with more buffer overflows performs worse than the other one.

3.5 Summary

In this chapter we introduced the SCORE operator which is an enhanced Eddy [AH00]. We extended Antoshenkov’s competition model [Ant93a] for joins and integrated it into SCORE. Therefore, we introduced Operator Groups (OpGrps) as new logical operators for SCORE. An OpGrp consists of two join operators which are running competitive. SCORE monitors the performance of both joins of a group and terminates the worse one after a certain time. Terminating an OpGrp implies merging the states of the two joins, i.e. moving the internal buffered tuples and calculating the missing join result. In order to avoid duplicates, SCORE partitions the input over the both competing operators appropriately.

Additionally, we introduced the complementary Merge-Hash-Join-Algorithms. We extended Operator Groups and SCORE for complementary joins. SCORE applies a different tuple routing strategy for complementary OpGrps in opposite to competing OpGrps. Instead of partitioning the input it is duplicated for both joins of the complementary group.

---

*To terminate a groups means to shut down one operator, move the tuples to the other and let the remaining operator calculate the remaining result.

*Input buffers are not to be mixed up with the internal buffers of operators.
The SCORE is also enhanced to deal with rejected tuples. Complementary OpGrps are designed such that the first operator of the group can reject tuples. SCORE sends these rejected tuples to the second operator of the complementary group.

As we want to use the blocking Sort-Merge-Join in Stratosphere, we included support for blocking joins into SCORE. SCORE and all blocking join operators are implemented in a (data-)partition-aware manner. SCORE routes the tuples partition-wise to the blocking operators and the blocking operators perform the join for each partition pair in order to calculate the result.
Chapter 4

Evaluation

4.1 Implementation

We implemented a standalone prototype of our system using the Java programming language version 1.6. The basic design of a SCORE system consists of four main parts: input streams, an output stream, query operators, and the SCORE operator. The prototype is implemented in a multi-threaded fashion. All input streams and operators as well as SCORE and the output stream are running in their own threads. Additionally, we run Group-Monitors (which will be explained later) within SCORE. Each monitor is running in its own thread as well. It is also possible to implement some query operators multi-threaded. Especially symmetric join algorithms can be implemented multi-threaded. As this is a prototype, we did not do a multi-threaded implementation of the symmetric join algorithms. We focused on the competition model within SCORE. But as the system is build modular, the current join implementations can be replaced with multi-threaded versions easily.

As input streams we implemented FileInputStreams and InMemoryInputStreams. FileInputStreams read the data from a file and send it to SCORE. InMemoryInputStreams buffer all data in main memory before sending the data to SCORE. The data is read from a file as well but this input step is done before we start query execution. InMemoryInputStreams enable us in our experiments to avoid cost intensive I/O-operations. Hence, we can simulate an environment where the data does not come from disk but from another process for example. Similar to input streams we implemented FileOutputStreams and InMemoryOutputStreams. FileOutputStreams write the result directly into a file while InMemoryOutputStreams buffer the result in main memory until query execution is completed. Afterwards the result is written to disk.

We implemented a bunch of query operators including Select and Joins. For each join we implemented a (partition) blocking and a symmetric version. Hence, our system contains Nested-Loop-Join, Symmetric-NLJ, Bucket-NLJ, Symmetric-BNLJ, Hash-Join, Symmetric-HJ, and Sort-Merge-Join. Additionally, we implemented the complementary Merge-Hash-Join (MHJ) consisting of Optimistic-Merge-Join and Auxiliary-SHJ as described in 3.3.1. A Merge-Join implementation is included as well even if the MJ can only be used as a single operator within SCORE. If there are multiple operators, SCORE
cannot guarantee that the input is routed to the MJ in sorted order. We include the MJ
to our experiments to evaluate the performance differences of single MJ, SHJ, and the
complementary MHJ.

Each operator and the output streams have input buffers assigned. The size (in
tuples) of this input buffers is configurable (from 1 to unlimited). We changed the sizes
of the buffers in our experiments to study the influence on SCORE’s routing decisions and
overall execution performance.

The implemented SCORE operator uses a special internal tuple buffer. The buffer
is divided into two parts. (1st) A limited buffer for new arriving tuples from the input
streams and (2ed) an unlimited buffer for all other tuples. The unlimited buffer contains
intermediate result tuples which are returned by the single operators. We use a limited
buffer for the new arriving tuples to prevent the input streams flooding the system with
new data if the system is busy anyway. SCORE routes tuples from the unlimited operator
buffer favored as suggested from Hellerstein and Avnur [AH00]. If this unlimited buffer
is empty, SCORE processes new tuples from the limited buffer. The internal buffer for
intermediate result tuples is unlimited to ensure that operators can return tuples at any
time. Otherwise, the system might run into a deadlock. If an operator cannot return a
tuple to SCORE, it will block. Thus, the operator stops processing and does not consume
tuples from its input buffers. If all these input buffers from all operators are full, SCORE
cannot send tuples to any operator. Hence, the operators are waiting for SCORE to send
tuples from the internal buffer to it (in order to make space for the tuple to be returned
by the operators). At the same time SCORE waits for the operator to consume tuples
from their input buffers.

SCORE is an Eddy operator enhanced with the competition model for joins. There-
fore, we implemented Operator Groups (OpGrp). An OpGrp consists two join operators
which are running either competitive or complementary. SCORE is aware if the group’s
algorithms are running competitive or complementary and sends the tuples to the opera-
tors within one group accordingly. In detail, SCORE sends tuples alternating to the two
operators if they are running competitive. In the complementary case, SCORE replicates
each tuple and sends a copy to each operator of the group. As we want to terminate one
of the operators within a group SCORE has to monitor the OpGrps. Group-Monitors
which are running within SCORE can terminate groups at any point in time. Terminat-
ing a group implies shutting down one join operator and merging the state of it into the
remaining operator (i.e. moving the internally buffered tuples into the winning operator
and calculating the missing join result). Additionally, the monitors inform SCORE that
the OpGrp got replaced by a single join operator. To be able to monitor and terminate
multiple OpGrps in parallel we run an own Group-Monitor for each OpGrp which watches
exactly the assigned group.

---

1 The select operator and the output stream have one buffer. All join operators have two buffers—one
for each input.

2 In this context block is not related to blocking vs. symmetric (pipeline) operators. A symmetric
operator blocks if it cannot return tuples to SCORE.
4.2. EXPERIMENTS

Demo GUI
Additionally to the actual system we implemented a Graphical-User-Interface (GUI). The GUI allows us to set up job configurations. These job configurations can be stored and loaded for later reuse. Jobs can also be executed using the GUI. The main aspect of executing a job with the GUI is the monitoring feature. The GUI shows different states of single operators like busy, idle, or blocking. It also shows the number of tuples of internal buffers and overall statistics about the number of processed tuples (grouped by tuple type) for each operator. A screen shot of the working GUI is shown in the appendix (figure A.1).

4.2 Experiments
We run our experiments on an IBM P550:8204 model E8A with 8x Power6-CPU 3.5 GHz and a main memory of 128GB RAM. The system has a local RAID attached containing 4xDS3400 with 33 HD of 146GB each. The RAID is connected to the system via multiple 4Gbit-FC-Adapters. The system runs a Virtual I/O-Server (VIOS) with 2 logical partitions (LPAR). We run our experiments on one of these LPARs (lpar1). lpar1 is configured to have 16 virtual cores and 32GB main memory. The installed operating system is an AIX Version 6.1 (Kernel version 6.1.3.1 TL03). For code execution we used Java 1.6.0, IBM J9 VM (build 2.4 for AIX ppc64-64).

We executed each test four times and measured the execution time with system time-stamps for each run. We dropped the runtime of the first execution for each experiment. Dropping the first run is necessary as the first execution takes longer because the Java just-in-time compilation (JIT) happens in this run. The Java Virtual Machine (JVM) is basically a byte-code interpreter. Interpreted code executes slower than compiled code. The JVM has a JIT-compiler which compiles the parts of the program which are executed often. In the beginning of the first run of the experiments the JVM interprets byte-code. After a certain time JIT-compilation happens, adding a compilation overhead to this run. Hence, the first run executes slower than the following in which no JIT-compilation happens and compiled code is executed.\(^3\) In order to get valid runtime results we dropped the first run of each experiment. The evaluation graphs show the average value of the remaining three test executions.

We use the same configuration for the SCORE system in all experiments. This configuration is quite simple containing only two inputs streams for both input relations which are joined and the corresponding join operator (or group operator). Nevertheless, SCORE is able to calculate more difficult queries. We used a modified version of the TPC-H query Q3 in order to verify correct query execution. But we did not use this or other complex queries for performance measurements. We use a generated data set in our performance tests. Each tuple is a single integer number, and we perform an equi-join on the data.

4.2.1 Symmetric joins
In the first set of our experiments we compare the performance of the Symmetric-NLJ, Symmetric-Bucket-NLJ, and the competing execution of both. In order to simulate a cost-

\(^3\)To avoid JIT-compilation it is necessary to repeat the runs within a single program (in opposite to start the program multiple times). If each run starts an own JVM, JIT-compilation happens in all runs.
CHAPTER 4. EVALUATION

```
1 // delay by busy wait
2 int i = 0;
3 int j = 1;
4 while (delay != 0) {
5     if (++i > 8) {
6         ++j;
7         i = 0;
8     }
9     if (j > delay)
10         break;
11 }
```

Figure 4.1: Simulating cost-intensive user-defined join-function.

```
<table>
<thead>
<tr>
<th>DU</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>AVG</th>
<th>time per DU (in µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>943</td>
<td>935</td>
<td>933</td>
<td>937</td>
<td>0.0094</td>
</tr>
<tr>
<td>5</td>
<td>4655</td>
<td>4620</td>
<td>4566</td>
<td>4613</td>
<td>0.0092</td>
</tr>
<tr>
<td>10</td>
<td>9766</td>
<td>9637</td>
<td>9636</td>
<td>9679</td>
<td>0.0097</td>
</tr>
<tr>
<td>20</td>
<td>18999</td>
<td>19088</td>
<td>18863</td>
<td>18983</td>
<td>0.0095</td>
</tr>
<tr>
<td>50</td>
<td>46650</td>
<td>46589</td>
<td>46619</td>
<td>46619</td>
<td>0.0093</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AVG</td>
<td>0.0094</td>
</tr>
</tbody>
</table>
```

Figure 4.2: Runtime in ms of delay-loop (100.000.000 executions).

intensive user-defined join-function (as described in 3.1.1) we are using delay-units (DUs). A DU is a single execution of the code shown in figure 4.1. The value of the variable delay is the amount of DUs. We do not use a call of Thread.sleep(time) because it does not model the cost-intensive join-function appropriate. Instead it would alter the thread-scheduling behavior because a sleep-call might imply a context switch to another thread. Additionally, it was not possible to use a simple for-loop (or nested for-loops) with an empty loop-body because the Java compiler eliminates this code during code optimization. Hence, we implemented DUs by the busy-wait-loop shown in figure 4.1.

We evaluated the time a single DU consumes by running the delay-loop 100,000,000 times. We executed the loop for delay-values 1, 5, 10, 20, and 50 and repeated each run three times. The result table is shown in figure 4.2. The measured results are stable and a single DU takes a little less than 0.01µs. The stability of the measured results implies that the busy-wait-code scales linear with the number of DUs. That is important to make it easier to interpret the test results.

Both inputs consists of 50,000 tuples each in these experiments. In the first set of experiments we made sure that the result is always empty. An empty result enables us to compare the time consumption of the actual join-predicate evaluation. As parameter we use a distribution-factor which has a value between 0 and 1. The former value
4.2. EXPERIMENTS

means that all tuples are equal; the latter value that we have unique tuples as input. A distribution-factor of 0 implies that the SBNLJ builds hash-tables with exactly one bucket. If the distribution-factor is 1, the number of buckets is equal to the number of tuples.

SNLJ vs. SBNLJ with 0 delay-units

Figure 4.3 shows the runtime of the SNLJ, SBNLJ, and SCNLJ (competing execution of SNLJ and SBNLJ). We used a delay of 0 DUs. We altered the value of the distribution-factor (y-axis) and plotted the average runtime in ms (x-axis). The runtime of the SNLJ (blue) is invariant under the distribution-factor. That is an expected result. The SNLJ evaluates all tuples-pairs, and the data distribution does not affect the runtime. The runtime depends only on the number of input tuples and how many DUs are used (both parameters are unchanged in this experiment). The SBNLJ (orange) performs best with distribution-factor 0. It hashes all tuples into a single bucket and calls $JOIN(\ldots)$ a single time. With growing distribution-factor the number of buckets and $JOIN(\ldots)$ calls increases. The SBNLJ needs more time to perform the task. At distribution-factor 1 the runtime of the SBNLJ is more than three times bigger than the runtime of the SNLJ. In this case the SBNLJ hashes each value into an own bucket. Hence, it has to perform as many $JOIN(\ldots)$ calls as the SNLJ. The hashing overhead is maximized and the SBNLJ cannot save a single $JOIN(\ldots)$ call. When we ran both algorithms competitive (green), we applied our basic shut-down strategy, i.e. we divided the complete input over the two algorithms before we choose one. Additionally to the runtime we checked which algorithm
won the competition. For distribution-factors 0, 0.1, and 0.25 the SBNLJ won. In the other cases the SNLJ won. This result is also reflected in the runtime of SCNLJ which follows the corresponding runtime of the single execution of the winning algorithm.

**SNLJ vs. SBNLJ with 50 delay-units**

We repeated the above test with DUs 1, 5, 10, 20, and 50. The complete result of these experiments is included in the appendix (section A.2). Figure 4.4 shows the result for 50 DUs. Despite of an overall higher runtime of all three algorithms we see a similar result as for 0 DUs.\(^4\) The difference is that the competition model now makes a wrong decision for distribution-factor 0.5. The reason why the worse algorithm wins for distribution-factor 0.5 lies in the non-linear runtime growth of the two algorithms. The SNLJ processes the first tuples very quickly and gets slower as the internal tuple buffers grow. The tuples which are processed later have to be compared with more tuples and a single loop takes more time. For the SBNLJ it is more likely that a tuple creates a new bucket in the beginning of the algorithm. Creating a new bucket implies a loop over all buckets from the other hash table and the linking of the buckets. Hence, the SBNLJ cannot save $JOIN(.,.)$ calls at the beginning of query execution but has a hashing overhead. As longer the algorithm is running as fewer buckets are created and as more $JOIN(.,.)$ calls are omitted. When the hash tables are build up (almost) completely the SBNLJ saves most. It can hash new

---

\(^4\)All three algorithms have a higher runtime compared to the first experiment because we use 50 DUs instead of 0 DUs now.
4.2. EXPERIMENTS

tuples into the appropriate bucket and joins the new tuples with all tuples from all linked buckets of the other hash table. Hence, the lion’s share of processing time is spent building the result. In summary, the SNLJ starts with a high performance and slows down while the SBNLJ starts slowly but speeds up later on.

For the small distribution-factors 0, 0.1, and 0.25 this initial phase is very short as a small number of buckets is created. With a growing number of distinct values (factor 0.5) the number of buckets grows and the initial phase consumes more time. The SNLJ has a better performance at the beginning and SCORE choose it. The past-predicts-future assumption fails.

The numbers also show that the SCNLJ performs a little worse than the single execution of the SBNLJ if the SBNLJ wins. This overhead is expected because of running an OpGrp. SCORE does more work due to a more complex tuple routing algorithm. Additionally, shutting down an OpGrp produces some overhead. As we have just one OpGrp in the system, this overhead is reflected in overall query execution time. For distribution-factors 0.5 and 1, the SNLJ wins the competition. The overhead of the SCNLJ is very big in these cases. The reason is that the shut-down phase is longer in these cases. As the SBNLJ performs slower now, the competition phase takes longer. Hence, more tuples are processed and buffered internally by the loosing operator. Additionally, it is more cost intensive to move the tuples from the SBNLJ to the SNLJ than the other way round. The tuples are buffered in hash tables and it takes longer to access them in order to move them to the SNLJ.

We expect that the overhead disappears or at least gets smaller if more operators are used. The shut-down overhead should be hidden by real parallel execution as other operators perform useful work during the OpGrp shut-down phase.

SNLJ vs. SBNLJ with short competition

We repeated the above experiments with a more aggressive OpGrp shut-down strategy. We integrated support for OpGrp shut-down after a certain amount of tuples is routed to the OpGrp. We use a value between 0.05 and 1 and calculate the tuple threshold as product of the value and the number of input tuples (of one relation).\(^5\) Figure 4.5 shows the result with our most aggressive shut-down threshold of 5% for 0 DUs. SCORE’s OpGrp routing overhead as well as the shut-down delay is much smaller compared to the default shut-down strategy. SCORE even altered its competition decision for distribution-factor 0.25 and choose the better performing SBNLJ.

More results with a shut-down threshold of 1.00, 0.75, 0.5, 0.25 and 0.05 are shown in the appendix (section A.2).

4.2.2 Blocking joins

To evaluate how blocking operator perform within a competing Operator Group we run the blocking HJ, SMJ, and their competing execution. Once again we altered the data distribution from 0 to 1. As the HJ and SMJ perform much faster than the NLJ and BNLJ, we increased the size of both inputs to 10,000,000 tuples each. The corresponding ASCII files containing the data have a size between 15 and 30 MB. Each tuple is a single

\(^5\) 1 is the default shut-down strategy.
Figure 4.5: SNLJ, SBNLJ and competitive execution of both (shut-down after 5% of input; 0 DUs).

Figure 4.6: HJ, SMJ and competing execution of both.
4.2. EXPERIMENTS

integer which is stored as a string. With increasing distribution-factor more different numbers are used. Hence, the numbers are getting bigger and need more space in there string representation.

Figure 4.6 shows the runtime of the three algorithms. The performance of all three is stable over all distribution-factors. Neither the HJ’s nor the SMJ’s performance depends on the data distribution. The HJ and SMJ have about the same performance. Hence, their competing execution has a similar performance. The tests of the SNLJ and SBNLJ already showed that there is almost no overhead for running a competing OpGrp. As both algorithms have about the same performance, it does not matter which algorithm won the competition.

We re-run this experiment with different group shut-down thresholds. We used the same values as we did for the NLJ-tests: 1.00, 0.75, 0.5, 0.25, and 0.05. All results are shown in section A.3 of the appendix.

4.2.3 Complementary join

In section 3.3 we introduced the complementary Merge-Hash-Join (MHJ) algorithm. We executed the MHJ and compared the result to the MJ and SHJ. We use a sorted input to enable the MJ to work correctly. Both input relations have 2,000,000 tuples each and a tuple is again a single integer. Figure 4.7 shows the runtime in ms for different data distribution-factors—the result was always empty. The MJ (red) has the best performance as expected. The complementary MHJ (dark green) has the worst performance and needs about twice the time of the MJ. The idea of the MHJ is to apply a Merge-Join to an almost sorted input. Hence, the performance was expected to be about the same as for the MJ. We investigated into this issue. The main difference between complementary and competing OpGrps is the tuple distribution strategy. In the competing case the input is partitioned equally over both parts of the group. For complementary OpGrps the input gets replicated. We measured the time SCORE needs to replicate the input. We subtracted the time spent for duplicating the input and included it in figure 4.7 as MHJ(c) (light green). The figure shows that a big amount of time of the MHJ’s overhead is spent for data replication.

Additionally to the actual data replication SCORE performs twice as many routing operation in the complementary case compared to competing execution. Both operators of the OpGrp receive all input tuples. Hence, SCORE performs more routing work as for a single operator or a complementary OpGrp. Figure 4.8 shows the execution time of the three algorithms with subtracted tuple routing time. *Tuple routing time* refers to the time SCORE needs to send the tuples to the operator or OpGrp. For the HMJ the routing time includes the data-replication time. The numbers show that the MHJ’s overhead is smaller compared to figure 4.7. That proves that routing tuples to a complementary OpGrp is more expensive than routing tuples to a single operator. The HMJ has a lower adjusted runtime as the SHJ for four (out of five) distribution-factors. Hence, the routing and duplication overhead is severe. It seems that the MHJ itself performs better than the SHJ. The problem is SCORE’s routing and data replication overhead.
Figure 4.7: SHJ, MJ and complementary MHJ (without any ooo-tuples—empty result).

Figure 4.8: SHJ, MJ and complementary MHJ (without any ooo-tuples) [routing time subtracted].
4.2. EXPERIMENTS

Merge-Hash-Join with out-of-order-tuples

In the former experiments we used a sorted input because we run the MJ in order to compare its performance to the MHJ. But the idea of the MHJ is to apply it to almost sorted data. We repeated the experiment with an input which is not completely sorted. The MJ algorithm cannot calculate the correct result now and we exclude it from these experiments. We use a probability factor which indicates how likely it is for a single tuple to be out-of-order. We executed the experiments with ooo-probability of 0.1, 0.25, and 0.5 to see how the MHJ’s performance is influenced by ooo-tuples. Figure 4.9 shows the runtime of the SHJ and MHJ for ooo-probability 25%. (The other results are shown in subsection A.4.1 of the appendix.) The runtime of both algorithms is in the same range as in the experiments without ooo-tuples. Nevertheless, the ooo-tuples add a little more additional overhead. SCORE has to route rejected tuples now. The SHJ has a stable performance for all ooo-probabilities.

The MHJ’s overhead ooo-tuples produce is unexpected low. This result is somewhat encouraging as it might be possible to alter the SCORE’s tuple routing strategy to reduce the tuple replication overhead. As both join-algorithms access the data read only, it would be possible to duplicate just the meta-data (routing bits) of each tuple and work on a single data copy. This strategy should reduce the overhead and the HMJ might perform better than the SHJ.

Figure 4.9: SHJ and complementary MHJ (with 25% ooo-tuples).
Merge-Hash-Join with non-empty result

In this experiment we use input data which produces a result. We include the MJ like in the first MHJ experiment. Hence, the input data must be sorted. The size of the input relation is still 2,000,000 tuples. We run the test with distribution-factors 0.1, 0.25, 0.5, and 1.0. We omit distribution-factor 0 because the result would be the Cartesian Product. All three algorithms need more time for a smaller distribution-factor (figure 4.10). With a smaller distribution-factor we have more duplicates per relation. Hence, the result is bigger. Therefore, all algorithms need more time calculating the result. The actual joining part dominates the runtime in opposite to the experiments with empty result. For distribution-factor 1 the result is similar to the experiment with an empty result. The MJ performs best and the MHJ worst. If we subtract the data replication overhead, the MHJ(c) performs again better then the SHJ. For all other distribution-factor the MJ performs worse than the SHJ. The reason in the bigger output. The MJ’s performance advantage is that it can skip tuples if they do not have a joining partner. If the join results gets bigger, the MJ cannot skip as many tuples as if the result is small. The joining part—which is the same for both algorithms—dominate the runtime and the MJ’s performance advantage over the SHJ gets lost.

The complete result of the HMJ experiments with a non-empty result is shown in the appendix (subsection A.4.2).
Chapter 5

Conclusion

5.1 Summary

In this thesis we investigated a new adaptive query optimization technique. We enhanced Antoshenkov’s competition model for competing join execution and integrated it into Avnur’s and Hellerstein’s Eddy operator. We call this new operator SCORE (Stratosphere’s Coordinating Operator for Re-arrangements). SCORE additionally supports (partition) blocking operators to be able to use the blocking Sort-Merge-Join-Algorithm. The Stratosphere system has to deal with unknown user-defined join-functions. We introduced the Bucket-Nested-Loop-Join algorithm in order to improve the execution of join using such function. We also introduced the complementary Merge-Hash-Join algorithm. It consists of a Merge- and a Symmetric-Hash-Join. The idea is to apply a Merge-Join to an almost sorted input. SCORE is able to execute the complementary Merge-Hash-Join as well.

We implemented a standalone prototype of the SCORE operator using the Java programming language and evaluated the new system with different experiments. The experimental results show that SCORE is able to detect the better performing join algorithm. At the same time the overhead of running multiple join algorithms competitive is negligible as the system runs in a multi-core environment. Additionally, we avoid a waste of resources by partitioning the input data in a way such that both competing algorithms produce useful output. This data partitioning technique also avoids cost intensive duplicate handling as duplicates get avoided at all.

5.2 Open issues

There are still open issues which should be investigated further. First, the applied strategy for picking the better performing algorithm is quite simple. It would be interesting to know what other—more advanced—strategies can be applied within SCORE. This investigation might include a very detailed study about the complexity of the used join algorithms. The performance of joins can change during query execution because the joins internal state grows (i.e. more tuples get buffered internally). Hence, some algorithms process the first tuples very quickly but get slower during query execution. Other algorithms start slowly
but speed up later on. It is an open question if we can describe the algorithms’ behavior in detail and if this knowledge leads to a better competition strategy.

It might also be possible to advance SCORE’s shut-down strategy if it cannot decide what algorithm performs better. In the current implementation operator number two of the group is terminated in such a case.

We added support for (partition) blocking operators. SCORE’s current lottery scheduling strategy does not put these into account. It is an open question how the tuple scheduling strategy should be altered if blocking operators are in the system. Additionally, it might be possible to weaken the blocking behavior of these algorithms by splitting the input into more partitions. This strategy pushes the algorithms into the direction of a pipeline execution. It is not straightforward to see, how such a partition-pipeline execution influences the performance of the system. The approach of using more partitions could help to improve the tuple routing strategy without a need to adopt it for blocking operators. Blocking operators would perform more small working steps instead of a big one. Hence, the ticket counts of the operators are updated in each step.

The performance of the complementary Merge-Hash-Join does not meet our expectations. The overhead of the data replication strategy is too big. We expected the MHJ to have a performance close to the MJ if little out-of-order-tuples are in the input. It would be interesting to see if the tuple replication strategy can be improved such that the expected performance is accomplished. SCORE replicates each tuple right now (including the meta-information for routing). As join algorithm accesses the data read-only, it would be possible to duplicate only the meta-information and work with a single copy of the actual tuple. It might also be possible to avoid replication completely by altering the Optimistic-Merge-Join and Auxiliary-Symmetric-Hash-Join. The reject strategy of the OMJ could be altered such that rejected tuples are not sent back to SCORE. Instead, the OMJ notifies the ASHJ about rejecting a specific tuple. The ASHJ has a copy of this tuple anyway and could use it to perform the join.

Additionally, it would be interesting to investigate if there are other complementary join strategies which could be applied within SCORE.

The next step to continue this work might be the integration of SCORE into Stratosphere. It would be interesting to see how SCORE performs within the system if it runs in a compute cloud with bigger resources and input data. Re-ordering PACT operators within the data-flow program is not trivial as the PACT semantic is more general than QEPs. But applying different competing join algorithms within Stratosphere’s second-order functions could improve overall query execution time.

As Stratosphere runs in a cloud, the PACT programs are executed in a highly parallel manner. The execution of a PACT program can involve hundreds or even thousands of machines. Hence, it seems to be difficult to use the centralizing SCORE operator. The idea of distributed Eddie from Tian and DeWitt [TD03] could be applied to SCORE to make it easier to integrate SCORE into Stratosphere.
Appendix

A.1 Graphical User Interface of the SCORE prototype

Figure A.1: Demo GUI monitoring a running SCORE system.
A.2  SNLJ, SBNLJ and their competing execution

A.2.1  Group shut-down factor 1.00 (default shut-down strategy)

Figure A.2: SNLJ, SBNLJ and their competing execution with shut-down factor 1.00.
A.2. SNLJ, SBNLJ AND THEIR COMPETING EXECUTION

A.2.2 Group shut-down factor 0.75

Figure A.3: SNLJ, SBNLJ and their competing execution with shut-down factor 0.75.
A.2.3 Group shut-down factor 0.50

Figure A.4: SNLJ, SBNLJ and their competing execution with shut-down factor 0.50.
A.2.4 Group shut-down factor 0.25

Figure A.5: SNLJ, SBNLJ and their competing execution with shut-down factor 0.25.
A.2.5 Group shut-down factor 0.05

Figure A.6: SNLJ, SBNLJ and their competing execution with shut-down factor 0.05.
A.3 HJ, SMJ and their competing execution

Figure A.7: HJ, SMJ and their competing execution.
A.4 SHJ, MJ and complementary MHJ

A.4.1 Empty result

Figure A.8: SHJ, MJ and complementary MHJ with empty result.
A.4. SHJ, MJ AND COMPLEMENTARY MHJ

A.4.2 Regular result

(a) Out-of-order-probability: 0%.

(b) Ooo-prob: 0% (subtracted routing time).

(c) Out-of-order-probability: 10%.

(d) Out-of-order-probability: 25%.

(e) Out-of-order-probability: 50%.

(f) Out-of-order-probability: 100%.

Figure A.9: SHJ, MJ and complementary MHJ with regular result.
Bibliography


Selbständigkeitserklärung


Matthias Sax

Einverständniserklärung
Ich bin damit einverstanden, dass die vorliegende Arbeit mit dem Titel “Dynamic Join Algorithm Switching at Query Execution Time” in der Bibliothek ausgelegt wird.


Matthias Sax