Main Memory Tuning of Stratosphere

Diplomarbeit

zur Erlangung des akademischen Grades
Diplominformatiker(in)

HUMBOLDT-UNIVERSITÄT ZU BERLIN
MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT II
INSTITUT FÜR INFORMATIK

eingereicht von: Zheng Wang
geboren am: 1982.04.13
in: People’s Republic of China

Gutachter(innen): Prof. Johann-Christoph Freytag, Ph. D.
Prof. Dr.-Ing. Ulf Leser

eingereicht am: ...... verteidigt am: ......
Contents

1 Introduction 1
   1.1 Memory Tuning and Database 2
   1.2 Conventions 3

2 Basic Concepts and Related Work 4
   2.1 Programming Model in Stratosphere 4
      2.1.1 Parallelization Contracts 4
   2.2 Join Implementation in Stratosphere 7
      2.2.1 Nested Loop Join 7
      2.2.2 Sort Merge Join 8
      2.2.3 Hash Join 9
      2.2.4 Reduce and Combine 10
      2.2.5 DataSink 10
      2.2.6 Summary 10
   2.3 Related Work 11
      2.3.1 Memory Tuning for Shared-Memory Multiprocessors 11
      2.3.2 Memory Tuning for Work Area 12
      2.3.3 Memory Tuning for Buffer Pool 13
      2.3.4 Memory Tuning based on Workload 15
      2.3.5 Memory Tuning based on Memory Consumers Behavior 16
   2.4 Return on Consumption Model 17
   2.5 Memory Management based on Memory Utilization 19
      2.5.1 Memory Utilization of Operators 20
      2.5.2 Memory Competing between Operators 24

3 Memory Tuning for Stratosphere 27
   3.1 Problem Formulation 27
      3.1.1 Goals of Memory Tuning 27
      3.1.2 System Architecture of Memory Tuning 27
      3.1.3 Problem definition 29
   3.2 ROC Estimation 30
   3.3 Asynchronous vs. Synchronized Memory Tuning 32
   3.4 Memory Tuning Algorithms 35
      3.4.1 Memory Competition Algorithm Based on Memory Utilization 35
      3.4.2 Summary 37
   3.5 Memory Adaptive Joins 37
      3.5.1 Workload Discovery 37
      3.5.2 Cycles for Adapting Memory 38
List of Figures

1.1 Three-step cycle of feedback performance tuning [WMHZ02] . . . . . . . . . . . . . 2
2.1 Implementation of join in MapReduce. . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.2 Operators of PACT Program [HPS+12] . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.3 Response time with different size of work area . . . . . . . . . . . . . . . . . . . . 12
2.4 SBPX and simulated buffers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.5 Scheduling policy for workload classes . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.6 Example for memory consumption and allocation curve . . . . . . . . . . . . . . 18
2.7 NLJ vs. BNLJ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
2.8 The memory cycling and data flow of sorter . . . . . . . . . . . . . . . . . . . . . . 22
2.9 The IO cost of sorting with different number of runs . . . . . . . . . . . . . . . . 23
2.10 Partitioning and buckets building by Hash Join . . . . . . . . . . . . . . . . . . . . 24
2.11 Word count example of Reduce . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
3.1 PACT Program of TPC-H Query 14 . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
3.2 Nephele DAG of TPC-H Query 14 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.3 The architecture of the memory tuning . . . . . . . . . . . . . . . . . . . . . . . . . 30
3.4 Workflows of different memory tuning techniques . . . . . . . . . . . . . . . . . . . 33
3.5 Example of Asynchronous Memory Tuning . . . . . . . . . . . . . . . . . . . . . . 34
3.6 An example of ROC table . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
3.7 The execution, memory tuning, and reconfigure cycle . . . . . . . . . . . . . . . . 38
3.8 Examples of FPF Replacement Algorithm . . . . . . . . . . . . . . . . . . . . . . . 39
4.1 The solution of Stratosphere . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
4.2 The solution of memory tuning for Stratosphere . . . . . . . . . . . . . . . . . . . 43
4.3 Performance Test of Nested Loop Join . . . . . . . . . . . . . . . . . . . . . . . . . 45
4.4 Performance Test of Block Nested Loop Join . . . . . . . . . . . . . . . . . . . . . 46
4.5 Performance Test of Hash Join . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
4.6 Performance Test of Reduce . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
4.7 Performance Test of Sort Merge Join . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.8 Performance Test of CoGroup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
4.9 PACT Program of TPC-H Query 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
4.10 Reduce Operator in TPC-H Query 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
4.11 PACT Program of TPC-H Query 7 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
4.12 Sort Merge Join in TPC-H Query 7 . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
4.13 Small scale Sort Merge Join in TPC-H Query 7 . . . . . . . . . . . . . . . . . . . 52
4.14 Hash Join in TPC-H Query 14 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
4.15 Reduce Operator in TPC-H Query 14 . . . . . . . . . . . . . . . . . . . . . . . . . . 54
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.16</td>
<td>PACT Program of TPC-H Query 19</td>
<td>54</td>
</tr>
<tr>
<td>C.1</td>
<td>Example of Block Nested Loop Join in Query 7</td>
<td>67</td>
</tr>
<tr>
<td>C.2</td>
<td>Example of Hash Join CNN in Query 7</td>
<td>67</td>
</tr>
<tr>
<td>C.3</td>
<td>Example of Sort Merge Join SCNN in Query 7</td>
<td>68</td>
</tr>
<tr>
<td>C.4</td>
<td>Example of Hash Join OL in Query 7</td>
<td>68</td>
</tr>
<tr>
<td>C.5</td>
<td>Example of Reduce in Query 7</td>
<td>69</td>
</tr>
<tr>
<td>C.6</td>
<td>Example of small scale Hash Join OL in Query 7</td>
<td>69</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Basic algorithm implemented in Stratosphere, their complexity and other details. . 11
3.1 Performance comparison of replacement algorithm . . . . . . . . . . . . . . . . . . . 39
Chapter 1

Introduction

In 2004, Google unveiled their MapReduce framework for big data parallel computing [DG04]. Google claims, that they processed over 20 petabytes of data per day by MapReduce framework in 2008 [DG08]. The MapReduce framework initiated a competition during which other big data analysis systems have emerged, such as Asterix/Hyracks [BCG11] from the University of California, Hadoop [SKRC10] from Apache Software Foundation and the Scope [ZBW12] from Microsoft.

The goal of this thesis is to conceptually develop a main memory self-tuning algorithm in the Stratosphere system [WK09, AHM10, BEH10, TUBP14]. Stratosphere is a platform for big data analytics using data parallelism. The memory configuration of data analytic platform without memory self-tuning has to be set, i.e. the setting a fixed memory amount to allocate for a specified operator will have to be done manually. When there are more applications built on the database, the complexity to find an optimal global setting for a kind of database operators is increasing. The more diverse a workload becomes, the more conflicts are conducted. It becomes unrealistic to find a fixed optimal memory allocation policy for a changing workload.

Memory tuning [WKKS99, CN07] tackles this problem by providing services to coordinate the memory allocation between memory consumers at runtime. The competition mechanism is introduced into the memory management. The memory allocation by memory tuning does not depend on fixed memory allocation policy setting but depends on the performance information collected at runtime. The performance information hints where the memory resource should be invested. Therefore, the system can adapt its memory layout to the workload automatically.

Big data analytic platforms like Stratosphere distribute their workload onto many compute nodes. The hardware of compute nodes can vary from each other. How the workload is distributed into multiple nodes is also unpredictable. A fixed memory allocation policy is not suitable in this setting as it can only be optimal for a specific workload distribution and a specific set of compute nodes. To adjust the memory allocation policy for all compute node from time to time is time consuming and not feasible. Therefore, memory tuning is suitable for Stratosphere to adapt the memory allocation policy to the hardware and the workload on all compute nodes.

In this thesis, we investigate the memory tuning technique to optimize the efficiency of memory usage. We introduce an asynchronous memory tuning (AMT) technique to allow the system to exchange the atomic units of main memory among operators without the time cost to synchronize or interrupt them. Unfortunately exchanging the memory between two operators directly requires the system to stop their execution and backup all intermediate results in the exchanging memory pages. The benefits of the memory tuning can be set off by the time cost itself and sometimes the memory tuning may have negative effect. AMT reduces the time cost of memory tuning itself. To quantify the memory utilization we use Return on Consume (ROC) [YC93] metrics. Based on the ROC value, we decide which operator should get more memory pages.
Design of join operators does not consider the memory changing at runtime. They are not required to adapt the changing memory allocation but implement certain join algorithm within fixed memory amount. Therefore, we adapt existing implementations of join operators for adaptive memory allocation. Because the AMT does not force the operators to stop the operators can decide when to react to changing memory allocation itself. We insert memory tuning points into operators where the operator can change its memory allocation with considerable time cost.

1.1 Memory Tuning and Database

![Three-step cycle of feedback performance tuning](figure1.1)

Today’s database management system DBMS rapidly grows in scale and complexity. Modern services require database management systems to provide a high performance availability and fault tolerance environment. Database administrators are responsible for the performance tuning. However, databases are in common use all over the world. Performance tuning in a runtime circumstance is complex and non-trivial. The database management system is required to tune itself according to the demand/workload and performance of applications and the hardware capabilities. However, due to the complex nature of DBMS, integration memory tuning algorithms into DBMS is difficult. However, several approaches have targeted this goal.

Feedback performance tuning is a cycle (Figure 1.1) in three steps: observation, predication, and reaction [WMHZ02]. The observation step monitors the performance of the database system. The parameters of workload, the throughput of queries, or the execution time of transactions can be viewed as indicators of system performance. These indicators are used to investigate whether the workload patterns have changed. During the prediction step the performance of tuning components in the next running period is forecasted by mathematical model. The result of prediction is used to optimize the setting of tuning components. The quality of prediction is crucial to the performance of system tuning. The prediction helps the system to choose the most effective adjustment among all tuning possibilities. There is a trade-off between various aspects. Improving one aspect introduces a risk of decreasing the performance of others. The system tuning reconfigurations should also consider the benefits of new configurations and the damage bought by shaking the stability. The prediction provides quantitative adjustments to system setting parameters, and the adjustment is carried out in the reaction step. The adaptable run-time mechanisms are necessary and crucial to
the system’s self-tuning. It is not easy to adapt new parameters while the system is serving its workloads. The system might be interrupted or synchronized to the request of parameter changes. The affects of parameter changing will be observed in the next system tuning cycle.

Memory management is one of the most important part of a database system. However, the workloads of applications grow much faster than the size of memory equipped in database systems. It is still crucial to improve the performance of database systems by reconfiguring the memory setting of memory consuming operators.

1.2 Conventions

We are using the following conventions in this work:

- We use capital litters starting with $R$ for dataset and input streams (e.g. $R$, $S$, $T$ etc.). In this work we access a dataset via input stream, therefore the dataset $R$ also indicates the associated input stream.
- The number of records in a dataset $R$ is referenced as $|R|$.
- The size of dataset $R$ in pages is referenced as $\langle R \rangle$. In this work, memory is divided into page frames, so we measure the memory size of a dataset also in the page frames.
- We use lower case letters for records. The letter corresponds to the dataset the records belongs to, i.e. record $r$, $r_1$, $r_2$ belong to dataset $R$.
- If two records are $r$ and $s$ are joined, the result record is $<r, s>$. 

The remainder of this thesis is structured as follows: Chapter 2 covers the basic concepts and related approaches used in this work. Chapter 3 presents our memory tuning approach AMT in Stratosphere and the detailed design of memory tuning operators. Chapter 4 describes the implementation and evaluation of our work. Finally, Chapter 5 presents our conclusions.
Chapter 2

Basic Concepts and Related Work

2.1 Programming Model in Stratosphere

We discuss the programming model of Stratosphere Parallelization Contracts (PACT) [AHM+10, BEH+11] in this section. PACT is a generalization of the well known programming model MapReduce [DG04].

2.1.1 Parallelization Contracts

The MapReduce programming model [DG08] simplified parallel data processing. In the MapReduce framework, the users express their queries with only two functions: Map and Reduce. The MapReduce system hides the aspects parallelization, fault tolerance, and workload balance from the user. The user can program a query in sequential code, and the execution engine distributes the workload into multiple computing nodes automatically.

However, MapReduce does not support binary operators like joins natively. Both Map and Reduce functions take a set of key/value pairs and a user defined function (UDF) as input. A native join operator should have two datasets and a UDF as input. To implement a join in MapReduce framework, the user has to create a union of two sets into one. In Figure 2.1, the Map function maps two sets into a union and marks records from R with color grey to identify which set they are coming from. The Reduce computes a Cartesian product of the union by UDF, and only when two records are marked with different colors, the Reduce emits a join result. The simplicity of MapReduce is thereby broken by the complexity of implementation of joins. The joins in user code are difficult to be detected by the execution engine. Joins are expensive and resource consuming operators. If the execution engine cannot detect joins, the execution engine also cannot assign resources properly to them. The optimality of the join depends on the implementation done by the user.

The PACT generalizes the MapReduce by introducing input-contracts. Every operator in PACT is expressed by combinations of input-contracts. Each input-contract describes an independent parallel data procedure. An input-contract is a second order function, which contains a user defined first order function and one or two datasets as input. The parallelization strategy of input-contracts is similar to the MapReduce system. Each call of the user defined second order function can be called independently from each other, so the multiple parallelized thread can be started to process the datasets in an identical way.

Definition 2.1 A first order function is a function that takes only values as arguments and returns values as result.
2.1. PROGRAMMING MODEL IN STRATOSPHERE

Figure 2.1: Implementation of join in MapReduce.

Figure 2.2: Operators of PACT Program [HPS+12]

Definition 2.2 A higher order function takes at least one lower order function as arguments or returns a lower order function as result. A second order function takes one or more first order function as arguments or result.

A PACT program is compiled into a Nephele directed acyclic graph (DAG) [AHM+10]. In the DAG, a vertex is an instance of database operator, and an edge is a channel. Each vertex presents a parallelized instance of database operator according to the input-contract. A channel transports data between instances of database operators. An input-contract can have one or two datasets as input. The corresponding instance of operator also has one or two channels as inputs. Functions from MapReduce are rebuilt into input-contracts, e.g. Map, Reduce and Combine. In Stratosphere we also define some input-contracts with two datasets as input to implement a join operator. Formally, we define $R$ and $S$ as two input datasets and $f$ is the user defined first order function.

Map The Map contract (Figure 2.2 a) is used to independently process each key/value pair. The input dataset is distributed in partitions, and each pair appears only once in one partition.
Inside of an instance of Map, there is an iteration to read the pairs and execute the UDF independently. The Map is defined as

\[ \text{Map: } R \times f \rightarrow f(r_1, r_2, \ldots, r_n) \].

**Reduce** The Reduce contract (Figure 2.2 b) groups input records by their keys. All pairs with the same key are grouped and merged by the same instance of the UDF. In an instance of Reduce, there is a sorter to group the pairs and the UDF is called independently for each group. The Reduce is defined as

\[ \text{Reduce: } R \times f \rightarrow [f(r_{k1,1}^1, r_{k1,2}^1, \ldots, r_{kn,1}^1), f(r_{k2,1}^2, r_{k2,2}^2, \ldots, r_{kn,1}^2), \ldots, f(r_{km,1}^m, r_{km,2}^m, \ldots, r_{kn,m}^m)] \],

where a set of attributes Key from record \( r \) is called key, the active domain of Key in \( R \) is \{\( k_1, k_2, \ldots, k_m \)\}. For the record \( r_k^k \) holds \( r_k^k, \text{Key} = k \).

**Combine** The Combine contract can improve the Reduce significantly. Similar to the Reduce, the Combine groups the key/value pairs by their keys and merges them. The difference of Combine and Reduce is that the Combine does not require that pairs with the same key in a partition are grouped in one set.

The single input-contracts are inherited from the MapReduce framework. Similar parallelization strategy is used to execute the single input-contracts. However, the parallelization strategy for binary input-contracts are difficult. When an input-contract has only one dataset as input, and the UDF is independently called for each pair, one pair is partitioned in exactly one partition. When there are more than one dataset, say two datasets \( R \) and \( S \) as input, the partition algorithm is more complex. The input datasets \( R, S \) are partitioned into parallelization units (PU). A PU is a portion of the input data from both datasets that is necessary for a call of UDF. For example, for an instance of Hash Join operator, the input data is partitioned into PUs according their hash code, e.g., the records with same hash code are partitioned into the same PUs. To compute a Cartesian product, one dataset is partitioned into an instance of operators, and the other dataset is broadcasted to all instances. The partitioning strategy is defined by the input-contract. In the following, we present an initial set of binary input-contracts of PACT.

**Cross** The Cross (Figure 2.2 c) contract has two input datasets. It produces the Cartesian product over both datasets. The UDF is called for each element in the Cartesian product. In an instance of Cross operator, there is a Nested Loop Join implemented. Formally, Cross is defined as

\[ \text{Cross: } R \times S \times f \rightarrow [f(r_1, s_1), f(r_1, s_2), \ldots, f(r_n, s_m)] \].

**Match** The Match contract (Figure 2.2 d) implements an equi-join between two datasets. Two pairs from both datasets are matched, when their set of key attributes are equal. The UDF is called for each matched two pairs. In an instance of Match operator, a Sort Merge Join or a Hash Join is used to find the matched pairs.

\[ \text{Match: } R \times S \times f \rightarrow \{f(r, s) | r, \text{Key} equals s, \text{Key} \} \].

**CoGroup** (Figure 2.2 e) The CoGroup contract partitions the key/value pairs of all input datasets according to their keys. Over all inputs, the pairs with the same key is grouped together and sent to the UDF. Although the CoGroup is defined to take arbitrary input set, to simplify the implementation, the CoGroup takes two datasets. More datasets can be expressed by chain of CoGroup contracts. In an instance of CoGroup operator, unsorted datasets are firstly sorted, then both sorted datasets are grouped by their keys.

\[ \text{CoGroup: } R \times S \times f \rightarrow [f(r_1^1, \ldots, r_{n1}^1, s_1^1, \ldots, s_{m1}^1), \ldots, f(r_1^{vl}, \ldots, r_{n2}^{vl}, s_1^{vl}, \ldots, s_{m2}^{vl})] \],

where \( r^v \) has \( r, \text{Key} = v \) (analog \( s^v \) has \( s, \text{Key} = v \)).
Output-contracts are optional contracts to describe the semantics of UDFs in the input-contracts. There are four output-contracts: same-key, super-key, partitioned-key and unique-key [AHM+10]. This information helps the execution engine to improve the data flow. For example, if a dataset is annotated with unique-key, the grouping and partitioning can be skipped in some cases. There are three types of channels: network, in-memory and file [AHM+10]. Each type of channels indicates the input data from which kind of media it is read. Channels are the edges in the Nephele DAG, the organization of channels indicates the how a PACT program is distributed. Data from in-memory and file allows random access, which allows operators to seek the data input to skip some unnecessary reading. However, the network channel is unseekable. To simplify the implementation of instance of operators, the type of channel is transparent to the user and the instance of operators. Instead of details of channels, Stratosphere takes all channels as an unseekable stream. This simplification makes it is easier for the system to organize the channels, but also entails some problem: the size of a channel is hidden behind the stream, so it is hard to predict how many records a channel will deliver. The performance estimation of instances of operators without hints of workload is therefore problematic. There are many memory consumers in the instance of operators. The instance of Cross is implemented by a Nested Loop Join. The instance of Match supports Hash Join and Sort Merge Join. The CoGroup also implements a Sort Merge Join. Another memory consumer is sorter. Stratosphere implements a Merge Sorter, which is implemented in Sort Merge Join, instance of Reduce, Combine and DataSink. We introduce a memory tuning algorithm to optimize the memory assignment to all memory consumers in chapter 3.

2.2 Join Implementation in Stratosphere

In this section we discuss the basic join algorithms used in Stratosphere. In this thesis, the memory utilization of joins is essential to the performance of memory tuning. In this section, \( R \) and \( S \) are the two datasets, which are joined. We set \( R \) as the inner and \( S \) as the outer. The function \( \text{join}(\cdot,\cdot) \) returns true if the two input records satisfy the join predicate - false otherwise. If a part of dataset \( R \) is loaded in a memory block, we use the \( \text{block}_R \) to indicate the block, and \( \text{load}(R, \text{block}_R) \) loads the next part of dataset \( R \) into the \( \text{block}_R \). The \( \text{load}(R, \text{block}_R) \) returns false if the input is exhausted, otherwise it returns true. All input dataset implemented in Stratosphere are in the form of stream. We use \( \text{next}() \) to traverse the stream. The \( \text{next}() \) returns the next record in the input stream, otherwise it returns the \( \emptyset \) if the stream is exhausted. The UDF used to build the join result is \( \text{emit(<r, s>)} \).

We present the worst case time complexity \( C_t \) and space \( C_s \) complexity of the algorithms. In the worst case the result of all joins could ever be the Cartesian product of \( R \) and \( S \). Here we use the number of \( \text{join}(r, s) \) a join algorithm needs to calculate the result. The worst case space complexities here is the space a join algorithm needs to run in cache. All joins we present here can calculate the join result with less space.

2.2.1 Nested Loop Join

In Stratosphere Nested Loop Join (NLJ) and Block Nested Loop Join (BNLJ) are implemented in Cross. NLJ reads every record from the \( S \) in a loop one at a time, passing each record to a nested loop that reads every record from \( R \). The NLJ produces a Cartesian product of \( R \) and \( S \). For each pair of \( <r, s> \) it output a join result (Algorithm 2.2.1).

The NLJ has a time complexity \( C_t \in O(|R| \cdot |S|) \). The space complexity is \( C_s \in O(|R| + 1) = O(|R|) \). The NLJ buffers only \( R \) in memory and it loads exactly one record from \( S \).
The BNLJ has a block of memory for buffering. It reads records from \( S \) into the buffer. For each record from the \( R \), it scans the buffer once. The BNLJ produces the same join results to NLJ. The BNLJ shows a better performance with less IO cost than NLJ by using additional memory to buffer a part of \( S \) in memory, when there is not enough memory to buffer all records of \( R \) in memory (Algorithm 2.2).

The NLJ has a time complexity \( (C_t \in O(|R| \cdot |S|)) \). The worst case space complexity is \( C_s \in O(|R| + 1) = O(|R|) \). When there is enough memory to buffer the dataset \( R \) into memory. The time complexity of BNLJ is \( (C_t \in O(|R| \cdot |S|)) \), The worst case space complexity is \( C_s \in O(|R| + |S|) \).

**Algorithm 2.1** Nested Loop Join Algorithm

1: function NestedLoopJoin\((R, S)\)
2:    for \((s \in S)\) do
3:        for \((r \in S)\) do
4:            if join\((r, s)\) = True then
5:                emit\((<r, s>)\)
6:            end if
7:        end for
8:    end for
9: end function

**Algorithm 2.2** Block Nested Loop Join Algorithm

1: function BlockNestedLoopJoin\((R, S)\)
2:    while load\((S, block\_S)\) = True do
3:        for \((r \in R)\) do
4:            for \((s \in block\_S)\) do
5:                if join\((r, s)\) = True then
6:                    emit\((<r, s>)\)
7:                end if
8:            end for
9:        end for
10:    end while
11: end function

### 2.2.2 Sort Merge Join

The Sort Merge Join (SMJ) sorts both datasets \( R \) and \( S \) then scans both datasets once to find qualifying records. The SMJ in Stratosphere is built in two phases: sorting phase and joining phase. In the sorting phase, both input datasets are sorted with quick sort. In the joining phase, both sorted datasets are scanned once. For each qualified record pair, the SMJ calls the second order UDF to emit the join results. Here, we assume that the input \( R \) is greater than \( S \).

The SMJ in the sorting phase, the SMJ sorts both input datasets \( R \) and \( S \) with quick sort [HPS08]. If an input is already sorted, say \( R \), sorting \( R \) can be skipped. In the pseudo code (Algorithm 2.3) the function \( sort(.) \) sorts one input dataset. The time complexity of the sorter is \( O(n \log n) \), whereby \( n \) is the number of records in the sorting input dataset. In the sorting phase the time complexity of SMJ is \( O(|R| \log |R| + |S| \log |S|) = O(|R| \log |R|) \). We need space to accommodate both datasets in memory for the sorting. The space complexity is therefore \( C_s = O(|R| + |S|) = O(|R|) \).

In the joining phase, SMJ performs an equi-join. SMJ outputs the join result when \( r = s \). SMJ will traverse both sorted datasets once. It compares the current records pair \( r, s \) from \( R, S \).
Algorithm 2.3 Sort Merge Algorithm

1: function SortMergeJoin\((R, S)\)  
2: \(R = \text{quicksort}(R)\) \(\triangleright /\text{sorting phase}\)  
3: \(S = \text{quicksort}(S)\) \(\triangleright /\text{joining phase}\)  
4: \(r = \text{next}(R)\)  
5: \(s = \text{next}(S)\)  
6: while \(r \neq \emptyset \land s \neq \emptyset\) do  
7: \quad while \(r \neq \emptyset \land r < s\) do \(r = \text{next}(R)\)  
8: \quad end while  
9: \quad if \(r = \emptyset\) then return  
10: \quad end if  
11: \quad while \(s \neq \emptyset \land r > s\) do \(s = \text{next}(S)\)  
12: \quad end while  
13: \quad if \(s = \emptyset\) then return  
14: \quad end if  
15: \quad if \(r = s \land \text{Join}(r, s) = \text{True}\) then  
16: \quad \hspace{1em} emit\((<r, s>)\)  
17: \quad end if  
18: \quad \end while  
19: end function

If \(r = s\), the SMJ outputs the Cartesian product of all consecutive records which are equal to \(r\) or \(s\). To simplify matters, we ignore the consecutive records here and in the pseudo code. If one record is smaller than another, say \(r < s\), we use the \(\text{next}(R)\) to load the next record of \(R\). The comparing is repeated, until one dataset is exhausted. The time complexity of the joining phase is \(O(|R| + |S|) = O(|R|)\). In the worst case, both datasets are sorted and kept in memory, then the space complexity is \(O(|R| + |S|) = O(|R|)\).

As a summary, the time complexity of SMJ is \(C_t = O(|R| \log |R|) + O(|R|) = O(|R| \log |R|)\) and the space complexity is \(C_s = O(|R|) + O(|R|)\).

2.2.3 Hash Join

The Hash Join (HJ) hashes one dataset \(R\) on the join attributes and builds a hash table with a UDF hash function, which is supposed to calculate the hash value in linear time. After the hash table has been built, \(S\) is scanned. For each record \(s\) from \(S\), \(s\) is probed into the hash table to find its join partners.

The HJ is executed in two phases: building phase and probing phase (Algorithm 2.4). In the building phase of the pseudo code, we use \(\text{hash}(.)\) to get the hash value of the record \(r\). The \(\text{build}(\text{hashvalue}, r)\) inserts the record \(r\) with its hash value into the hash table. For records, which have the same hash value, HJ keeps a list of all buckets for all keys. The time complexity of the building phase is \(O(|R|)\), and the space complexity is also \(O(|R|)\).

In the probing phase, we scan the probing side dataset \(S\) and use the same hash function \(\text{hash}(s)\) to get the hash value of the \(s\). The \(\text{probe}(\text{hashcode})\) finds all records from \(R\), and HJ joins \(s\) with every hit records from \(R\). In the probing phase, each record \(s\) is visited only once, so the time complexity of the probing phase is \(O(|S|)\).

In both phases, each dataset is scanned once with a linear time cost. The overall time complexity is \(C_t = O(|R| + |S|)\). It needs space to build the hash table. The \(S\) is scanned once without any space demand. The space complexity for HJ is \(C_s = O(|R|)\).


Algorithm 2.4 Hash Algorithm

\begin{algorithm}
\begin{algorithmic}[1]
\Function{HashJoin}{R, S}
\For {r ∈ R}
\State \texttt{hashvalue} = \texttt{hash}(r)
\State \texttt{build(hashvalue, r)}
\EndFor
\For {s ∈ S}
\State \texttt{hashvalue} = \texttt{hash}(s)
\State \texttt{T} = \texttt{probe(hashvalue)}
\For {t ∈ T}
\State \texttt{emit(< t, s >)}
\EndFor
\EndFor
\EndFunction
\end{algorithmic}
\end{algorithm}

2.2.4 Reduce and Combine

The Reduce groups the input records, and merges records in each group. Stratosphere uses a quick sort to group the input records. Input records with the same key attributes are grouped together. After the input record is grouped, it calls the UDF for each group.

The Combine is similar to the Reduce, but it is unnecessary to group all input records with the same key attributes in a partition in one group. The Combine is an optional operator before Reduce to reduce the size of input dataset. The UDF used in the Combine should be the same as Reduce.

The time complexity of Reduce and Combine is equivalent to a sort in SMJ \( C_t = O(|R| \log |R|) \) and the space complexity is \( C_s = O(|R|) \).

Reduce is not a join operator, but it has a memory consumer, we also consider the Reduce in memory tuning. Combine is a memory consumer, but it has no IO cost. The memory allocation does little on its performance, but bigger memory produces a better combining result. We use a fixed memory allocation for Combine.

2.2.5 DataSink

DataSink is an operator to extract result dataset from output datasets of other operators. It is also equipped with a quick sort to give the result dataset an ordering. Sorting is optional. The time complexity of DataSink is equivalent to a sort in SMJ \( C_t = O(|R| \log |R|) \) and the space complexity is \( C_s = O(|R|) \).

2.2.6 Summary

Table 2.1 summarizes the differences of the join algorithms. The table includes time and space complexity as well as the ability to Theta Joins and the precondition to perform the join. HJ has the best time complexity. Both SMJ and HJ have the limitation to work on equi-joins only. Although with SMJ it is possible to implement the theta-join with \(<, >, \leq, \geq\) normally, but Stratosphere allows us to traverse the output of a sorter only once, so the theta-join with \(<, >, \leq, \geq\) is not supported. To implement a theta-join in Stratosphere, a NLJ or BNLJ is needed, though the NLJ and BNLJ have the highest complexity. They can evaluate any join condition. The SMJ requires a total order on comparing attributes. If the attributes do not have total order, HJ should be the alternative join algorithm. Although HJ evaluates equi-joins with less complexity than SMJ, but the SMJ outputs sorted result records. When both datasets are sorted, the SMJ is transformed to
2.3. RELATED WORK

In this section we discuss the memory tuning approaches from other works. The memory tuning technique is implemented on the hardware for the cache tuning for Multiprocessor architecture. On the software level, there are many works investigate the memory tuning from different points of view. The self tuning of memory configuration is implemented for buffering, work area, and some one time short term operations. The methods used in different approaches are various. We can reconfigure the memory allocation by balancing the memory capacity of buffer pools, implement a smart scheduling algorithm, give an overall reconfiguration by memory level or feedback information, and so on.

### 2.3.1 Memory Tuning for Shared-Memory Multiprocessors

In this subsection we discuss the memory tuning technique installed on the hardware of shared memory multiprocessors. To improve performance and to reduce energy cost, ‘Memory-based Computing for Performance and Energy Improvement in Multicore Architectures’ [RMB12] introduced a genetic program to tune the L1 cache allocated by running memory based computing operations. ‘A Dynamically Reconfigurable Cache for Multithreaded Processors’ [SCGG06] gives a solution to reduce the overall memory latency with feedback techniques.

Generally, the shared memory multiprocessor architecture refers to a block of random access memory can be accessed by multiple different CPUs with associated caches via an interconnection network. In shared memory multiprocessing all CPUs share a unique memory addressing space. Cache is a smaller and faster memory component to reduce the average memory access time of CPUs. The CPU can read and write data in the cache. L1 caches are fast memory components associated to the CPUs and L2 caches are shared among CPUs. The cache can be partitioned into equal sections called ways. The smallest size of memory block can be loaded into the cache called cache line.

By tuning the cache configuration of feasible data and instruction memory allocation of operators at runtime can save both energy consumption and improve performance in the presence of deadline constraints [SCGG06]. The idea of their work is to reconfigure the size of L1 cache for data and instruction of running operators to estimate the execution time and the energy expenditure. The genetic algorithm (GA) generates solutions by reproduction, crossover and mutation. Solutions that do not satisfy the deadline constraints and memory demand of operators are removed to make sure all solutions are feasible. The GA stops when there a solution meets the criteria, and the new reconfigure parameter is performed to the tuning of cache allocation of operators.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time complexity</th>
<th>space complexity</th>
<th>theta-join</th>
<th>requires</th>
</tr>
</thead>
<tbody>
<tr>
<td>NLJ and BNLJ</td>
<td>$</td>
<td>R</td>
<td>\cdot</td>
<td>S</td>
</tr>
<tr>
<td>SMJ</td>
<td>$</td>
<td>R</td>
<td>\log</td>
<td>R</td>
</tr>
<tr>
<td>HJ</td>
<td>$</td>
<td>R</td>
<td>+</td>
<td>S</td>
</tr>
<tr>
<td>Reduce</td>
<td>$</td>
<td>R</td>
<td>\log</td>
<td>R</td>
</tr>
<tr>
<td>Combine</td>
<td>$</td>
<td>R</td>
<td>\log</td>
<td>R</td>
</tr>
<tr>
<td>DataSink</td>
<td>$</td>
<td>R</td>
<td>\log</td>
<td>R</td>
</tr>
</tbody>
</table>

Table 2.1: Basic algorithm implemented in Stratosphere, their complexity and other details.

a merge join algorithm, and the complexity is the same to HJ. Reduce is not a join algorithm, it aggregates records.

2.3 Related Work

In this section we discuss the memory tuning approaches from other works. The memory tuning technique is implemented on the hardware for the cache tuning for Multiprocessor architecture. On the software level, there are many works investigate the memory tuning from different points of view. The self tuning of memory configuration is implemented for buffering, work area, and some one time short term operations. The methods used in different approaches are various. We can reconfigure the memory allocation by balancing the memory capacity of buffer pools, implement a smart scheduling algorithm, give an overall reconfiguration by memory level or feedback information, and so on.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time complexity</th>
<th>space complexity</th>
<th>theta-join</th>
<th>requires</th>
</tr>
</thead>
<tbody>
<tr>
<td>NLJ and BNLJ</td>
<td>$</td>
<td>R</td>
<td>\cdot</td>
<td>S</td>
</tr>
<tr>
<td>SMJ</td>
<td>$</td>
<td>R</td>
<td>\log</td>
<td>R</td>
</tr>
<tr>
<td>HJ</td>
<td>$</td>
<td>R</td>
<td>+</td>
<td>S</td>
</tr>
<tr>
<td>Reduce</td>
<td>$</td>
<td>R</td>
<td>\log</td>
<td>R</td>
</tr>
<tr>
<td>Combine</td>
<td>$</td>
<td>R</td>
<td>\log</td>
<td>R</td>
</tr>
<tr>
<td>DataSink</td>
<td>$</td>
<td>R</td>
<td>\log</td>
<td>R</td>
</tr>
</tbody>
</table>

Table 2.1: Basic algorithm implemented in Stratosphere, their complexity and other details.

a merge join algorithm, and the complexity is the same to HJ. Reduce is not a join algorithm, it aggregates records.
CHAPTER 2. BASIC CONCEPTS AND RELATED WORK

2.3.2 Memory Tuning for Work Area

Work area is the memory used for database operators to process their inputs. For example, a sort operator needs memory space to perform an in-memory sorting, a hash operator needs memory to build hash tables. The memory allocated for work area is normally assigned to individual operators before the operator is created to guarantee that they have enough memory to process the input data. The size of memory allocation is determined by the requirement of database operators. Operators define the minimal memory to perform their algorithm. The size of the work area affects the performance of operators significantly. To assign limited memory resources to running operators is important to improve the global performance. A memory tuning mechanism to reconfigure the work area size at runtime based on Oracle database system is introduced in "SQL Memory Management in Oracle9I" [DZ02].

Three work area size levels are defined to their workloads (see Figure 2.3) [DZ02]. Cache size is the size to accommodate the complete input data and auxiliary structures in memory, so that the operator performs only in cache without IO cost. One pass is the size smaller than the cache size, so the one more pass is performed over all input data. Multi-pass refers to a smaller than one pass and bigger than minimal memory requirement. By multi-pass, the IO cost of the operator increases dramatically.

The memory tuning introduced based on Oracle database system is level-wise memory tuning to reconfigure the memory used for sorting and hash joins [DZ02]. The system computes a memory

A feedback technique can be also used to determine the cache allocation of threads [SCGG06]. By employing a Least Recently Used (LRU) replacement algorithm, two partition control mechanisms are presented in their work. The first partition control is a synchronous feedback system. There is a counter to record the number of evicted cache lines and get the average evicted count for a sampling interval. The reuse weight is recorded by multiplying the average. Then, threads can be competed with each other by their reuse weight. The threads with high reuse weight can get one or two more ways or share an additional way with low reuse thread. The second partition control regulates the LRU replacement policy. The system records the frequency of accessing of each cache line. If the frequency of a cache line is higher than a threshold, the cache line is not recommended to be evicted. This mechanism exploited a simplified Least Frequently Used (LFU) algorithm to assist the LRU.

Figure 2.3: Response time with different size of work area
2.3. RELATED WORK

bound to measure all memory allocation of operators. If an operator’s cache size is smaller than the memory bound, then only the cache size is assigned to the operator. If the memory bound is less than the minimal memory requirement of an operator, the minimal memory is the target memory assignment. If the memory bound is between the minimal and cache size, the memory bound is equal to the expected memory allocation. If a work area is parallel, the new memory allocation is multiplied by its degree of parallelism. It also announces that additional pages for sorting do not improve the performance of the sorting when the memory allocation is between cache and one pass. Therefore, the memory allocation for sorting operator in this case will be set as one pass size. The memory tuning component reconfigures the memory allocation for work area once in very several seconds.

The level-wise memory tuning [DZ02] gives a simple solution for memory tuning, but it has some weaknesses. It is hard for operators to react to the memory reconfigurations in the interval between two reconfigurations, because the resize takes time to back up loaded data, when an operator is required to size down. The sizes of the different levels are difficult to estimate, because the auxiliary data structure and fragmentation caused by page frame undermine the estimation.

2.3.3 Memory Tuning for Buffer Pool

The memory tuning for buffer pools is the most frequently discussed topic in memory tuning. Generally the aim of memory tuning for buffer pool is to reduce the missing rate by replacement algorithms. When a page is required but missing in the buffer pool, the system will discard a page and load the required page into memory, so one missed page cost one page IO read and write. The widely accepted idea is to keep the hot pages in the buffer pool to improve the replacement algorithm, usually LRU.

Fragment Fencing and Class Fencing

Two famous works in this area are ‘Managing Memory to Meet Multiclass Workload Response Time Goals’ [BCL93, Ten95]. Fragment fencing and class fencing are introduced to tackle the memory tuning for buffer pool. A fragment is a data file or a subset of file pages. The idea of fragment fencing is that, the system creates a local LRU (LLRU) for each fragment. Given a response time goal, the system adjusts the number of pages in LLRUs to meet the goal. If a fragment’s response time is under the goal, it can get more pages, conversely if a fragment’s response time is shorter than the goal, it will shift some pages for other fragments. In this model the memory allocation policy is towards fragments, which have the most hot pages. If the hot pages are divided from valueless pages of a set of file, the memory management for them has a higher efficiency. For example, the pages for a B-tree have different frequency of references. If we always search a record from the root to the leaf, the possibility of a node referenced becomes lower and lower. If we divide the B-tree into fragments by its level, the fragment closer to the root has higher hit rate. By sequential access, leaves are traversed only once in a short time, in most case leaves occupy all the pages allocated for the B-tree and the hot pages are evicted. Dividing a fragment into fragments builds a fence to protect the frequently referenced non-leaf node from the possibly one time using leaves.

The fragment fencing provides a solution of memory tuning from the point of view of data storage. It is widely accepted by following works, but the methods to divide a fragment depends on the design of data structure and has to be done manually. Class fencing provides a solution from the point of view of memory consuming classes, and balances the response time for classes regardless, to which fragment a page belongs to. In class fencing, a memory page belongs to a class or the global buffer manager. The system divides all classes into goal and no-goal classes. No-goal classes do not take part in the memory tuning. The global buffer manager manages some "less
valuable" pages and pages of no-goal classes. Given a response time goal, all goal classes compete with the global buffer manager. If a class cannot meet the goal, it robs a page from the global buffer manager when it has a buffer miss. If a class has too many pages, it gives up pages to the global buffer manager. The global buffer manager implements a replacement algorithm for no-goal classes.

Compared to fragment fencing, class fencing does not depend on the manual setting; the system can balance the performance of goal classes without knowing the semantic of classes.

Extension on Buffer Pool Replacement Component

Another famous memory tuning approach is implemented in DB2 [SGAL+06]. They designed a Simulated Buffer Pool Extension (SBPX) to allow memory self tuning (see Figure 2.4). SBPX simulates an extended buffer. The cost-gain analysis of SBPX is built on saved system time per unit memory. Both buffer pool and associated SBPX implement a replacement algorithm. The memory tuning program is invoked in a fixed time interval to adjust the memory configuration between buffer pools. When a page is victimized, the identifier of the page is stored in the SBPX and the page is replaced. The time cost of reading this page is timed as cumulative saved time (CST). CST is considered as the cost of not keeping the page in memory. If a page is not found in the buffer pool, the system consults the SBPX, whether the identifier of this page is stored in the SBPX. If the identifier is found in the SBPX, then victiming this page is considered as a mistake. The cost to reload this page can be saved, if this page was not victimized. The number of identifiers in this SBPX is the number of additional pages needed to save the time used to reload pages. The total CST for a local buffer pool is normalized by the identifier in the SBPX, since the maximal size of a SBPX is fixed. The more identifier in the SBPX means more pages are replaced in the interval but less hot pages in the SBPX during the tuning interval, so the normalized CST is lower. Compared with normalized CST of buffer pools, the system can find out which buffer pool should get more memory pages.

The idea of SBPX is a memory self-tuning design. It is independent from the semantics of data and memory consumers. The performance of buffer pools is the only speculation of the memory management.

Inspired by SBPX, 'Lightweight Performance Forecasts for Buffer Algorithms' [B11] gives a solution to upsize or downsize a buffer is presented. The upsizing simulates a larger buffer similar to SBPX. The downsizing uses a different logic. It simulates one or more smaller buffers. The idea of downsizing is to find out whether a hit on the actual buffer also happens in a simulated smaller buffer. If the size of hot pages fits into a smaller buffer, then the performances of the actual buffer and the smaller buffer are similar. In other words, the actual buffer suffers less by downsizing. There are estimated cost of upsizing and downsizing as well as the measured cost of current memory allocation available for the memory tuning. The system shifts pages from buffers, which suffer less by downsizing to buffers, which get more benefit by upsizing. Considering the performance improvement of a bigger buffer, the SBPX is an optimistic approach. Their work also considers the pessimistic side to find a smaller buffer’s performance. It is an extension of SBPX. Figure 2.4 shows the relationship between undersize and overflow pages.

Performance Forecasting by Reference

Because the reference sequence of memory pages are predictable, there are also works that try to use the reference sequence to predict the memory shortage in the short future of classes to reconfigure memory allocations. Given a reference sequence, buffer management algorithm, and the size of buffer, the number of page faulting in the future is computable. The estimation of faulting pages is used to decide the size of buffer and the replacement algorithm for different buffer pools [CR93].
2.3. RELATED WORK

In their work, they tune the memory allocation for tables queried by single queries. If a query is executed repeatedly, e.g., user views, the feedback helps to assign available memory to buffer the tables involved in the query. They measure the average marginal gain for each buffer pool. When a query requires pages from a buffer pool, the system traces the reference sequence and records the time cost. Using the recorded reference sequence, the system estimates the time cost of assigning only 1 page to the buffer pool. The average marginal gain of the buffer is defined as the time cost difference between current allocation to 1 page divided by the allocation difference between current size to 1 page. A marginal gain function is also figured out to find the maximal memory allocation. When the marginal gain equals 0, more memory cannot improve the performance, so the allocation should never exceed the memory allocation, whose marginal gain equals 0.

When a query is executed for the first time, the system uses an average memory policy to give memory equally to all buffers involved. After the first execution, the marginal gain and the maximal memory allocation is computed. From the second time, given an available memory to the query, the buffer is allocated proportional to their average marginal gain. After the execution, the average marginal gain and the maximal memory are updated.

This memory tuning approach has a limitation. The memory tuning decision cannot affect the buffer pool size for a long time. It allocates memory for every query and improves the performance of single queries. It lacks a global point of view.

There are also other memory tuning techniques introduced in [THTT08, TMP03] to self-tuning the buffer pools by missing rate.

2.3.4 Memory Tuning based on Workload

Another essential factor to memory tuning is workload. In a concurrent query environment, scheduling determines the present executing workload. Traditionally, the memory allocation only considers to allocate proper memory amount to queries. This memory policy ignores the global memory demand, so the memory tuning can also be performed by adjusting the sequence of queries. 'Dynamic Memory Allocation for Multiple-Query Workloads' [MD93] studies the relationship between the scheduling and the memory allocation policy. They classify the queries in three classes: small, medium and larger medium query classes. The small query class requires 5% of main memory as average memory allocation. The medium query class asks for 10% to 95% main memory. The large query class requires more memory than the physical memory available. The system can allocate memory to joins in three different policies, minimum, maximum and available. All classes can be executed with minimal memory. The small and medium query class can wait in the queue until the system allocates enough memory to fit their workloads into the memory by maximum memory policy. The queries in large class are impossible to use maximum allocation. By available policy, a query is given the available memory, if the minimal memory requirement is fulfilled. Their study suggested that for

![Figure 2.4: SBPX and simulated buffers](image-url)
small queries maximum policy is superior to other setting, because the memory allocation is quite small and the waiting time is very short. The large query class should use minimum policy. If large query use available memory, it occupies all memory and other queries are blocked.

By classifying the query class and the memory allocation policies, they designed and compared different scheduling policy. First Come First Serve (FCFS), Responsible and Adaptive scheduling are examined (Figure 2.5). FCFS is a simple scheduling policy, but it is also troublesome. FCFS with maximum allocation policy blocks small queries behind large and medium queries and the multi programming level (MPL) of medium queries is also limited. With minimal allocation policy, the FCFS launches too many queries. Responsible is biased in favor of the small queries. The small queries use FCFS scheduling policy. The medium and large queries use separate waiting queue to prevent them from blocking small queries. The small queries have the priority to get memory, and the rest is shared by the medium and large queries. The system allows three classes to have at least MPL of one to avoid starvation. The adaptive scheduling uses three separate queues to control the MPL of three query classes. All queues perform a first in first out strategy. When a query of a particular class is done, the next waiting query in the queue is allowed to allocate memory. The MPL of three query classes is dynamically adjusted by their performance. The class with poorer performance is rewarded by higher MPL and the MPL of other classes is decreased.

The 'Dynamic Memory Allocation for Multiple-Query Workloads' [MD93] provides a point of view to tuning the memory allocation between query classes by implementing different scheduling and memory allocation policy. It gives a solution how to balance the scheduling of different size of workloads. The system can dynamically decide the number of queries in executing and gives a appropriate memory allocation for them. The memory resource can be used efficiently by responsible and available schedule policy.

2.3.5 Memory Tuning based on Memory Consumers Behavior

Short-Term Memory Consumers

The short-term memory consumers (SMC) like any event in the database can require large amounts of memory and cause lots of IO operations for a short period time, e.g., sorting. Sorting is typically an one shot operation and the result is accessed by a single client only once, but can cause a huge amount of IOs to store and read its intermediate data. In the progress a same piece of data may be read sequentially many times in a short time interval. By LRU management, sort flushes many
recently used pages into disk and it can suddenly change the demand of buffer pool.

The short time consumers and other memory consumer can be managed separately by using two buffer pools [O11]. The buffer pool for SMC starts with no pages inside. When a SMC is invoked, the system estimates the IO cost of the SMC. If a separated buffer pool benefits the performance, the main buffer pool is scaled down and a certain number of pages are shifted into the new buffer pool. After the SMC is done, pages are returned back into the main buffer pool. Their memory tuning technique reduces the IO cost of SMCs by providing an individual memory buffer pool and keeps the hitting rate of the main buffer pool at a low level.

**Memory Consumers with UDFs**

User defined function is frequently discussed in database system topics. Multimedia, web based applications, text search, time series manipulations, and analysis often make extensive use of UDF with database. Some UDFs appears as a join predicate, which might be invoked more than one million times. In 'Self-Tuning Cost Modeling of User-Defined Functions in an Object-Relational DBMS' [BHL05], the work provides two methods to allocate and deallocate memory for the buffer pool by historical statistics of UDFs.

They suggest that the data region with a complex cost model should be divided into small data regions, and allocate more memory for more frequently accessed data regions. The data region with less complex data distribution and less frequently accessed data region should be aggressively compressed by memory deallocation.

The data region with a complex cost model implicates that in the region the costs are various between different parts, so using a smaller division in the data region allows the system to implement a refined algorithm for each division. This method implicates the conception of fragment fencing, and provides a method to self tuning the division of fragments. The data region with less complex data distribution should be deallocated aggressively because these region can archive a similar missing rate without using as much memory. In their study [BHL05], they use a partition and merge algorithm to insert divisions into the data region and then merge the similar regions to an entire partition by the histogram of the cost of UDFs.

The memory allocation and deallocation for the data region with access frequency is a kind of memory tuning method by the utilization of memory. Allocating more memory for frequently accessed regions helps to reduce the missing rate of the buffer pool.

### 2.4 Return on Consumption Model

Return on Consumption (ROC) was introduced by Philip S. Yu and Douglas W. Cornell in 1992 [YC93]. Conventionally, the memory allocation is determined by workload information of queries. In their investigation, they find that the marginal utilities of memory of queries are vastly different. The marginal utilities are the derivative of throughput/running time with respect to the memory allocation. Theoretically, the optimal memory allocation is achieved when marginal utilities of memory of all queries are equal. Therefore, the conventional memory management is suboptimal. They introduced the ROC model to estimate the efficiency of allocating additional memory to a query. In this thesis, we employ the ROC model to evaluate the effectiveness of memory allocations.

The memory affects the performance of memory consumers. Queries without a join operator require only little memory, so we consider joins in this section. The running time of a join operator can decrease or increase with the size of its allocated memory. The idea of ROC has two important parts: the return and the consumption. For a given join and two input datasets, we expect a shorter running time by assigning more memory to the join operator. A join operator can be set idle before it gets the first record, and this waiting time depends on the allocation strategy of the
system rather than the memory allocation. Therefore, here we define time cost $T$ as the time cost in this section as the time elapsed from the point that a join operator gets the first record to the point that the operator release all memory.

Assume that join $J_1$ has a bigger workload, while join $J_2$ has a smaller. If both joins allocate the same amount of memory and perform the same join algorithm on their workload, join $J_1$ has a longer time cost than join $J_2$. The memory allocation does not reflect the difference between $J_1$ and $J_2$. Hence, we define the memory consumption as a space and time product $MC = MT$, whereby $M$ is the allocation. The memory consumption is formally defined as an integral.

**Definition 2.3** *The memory consumption*

$$MC = \int_0^T F(t)dt$$

, whereby $F(t)$ is the instantaneous memory allocation.

In Figure 2.6 two examples of memory consumption and allocation curve are shown. Figure 2.6 a) presents a join that with increasing memory allocation has almost a linear effect on reducing the query time cost, e.g., a Hash Join. Figure 2.6 b) presents a memory consumption curve in stages. In some cases, allocating more memory cannot improve the performance of an operator, unless the additional memory reaches a certain amount, then the time cost is reduced suddenly, for example the Sort Merge Join. Sorting a dataset in memory or partially in memory has a significant difference. Assigning more memory to a sorter whose memory allocation is between one pass and in cache rarely reduces the time cost, but the time cost reduces suddenly when all data are accommodated in cache [DZ02].

In a shared memory architecture, the total memory consumption in a period is limited. The memory allocation with short running time and a high consumption is not always valid for all joins.
Join operators have a minimal memory allocation. This minimal value is not a native character of a join, but a setting given by the programmers. We set the minimal memory allocation $F_{\text{min}}$ as a reference point. Normally, additional memory improves the performance, otherwise it is unnecessary to assign more memory to this join without reducing the running time. For a feasible memory allocation $F$ the time cost is $T(F)$, $F > F_{\text{min}}$ and $T(F) < T(F_{\text{min}})$. The reduction of running time $\Delta T = T(F_{\text{min}}) - T(F)$, the additional memory is $\Delta F = F - F_{\text{min}}$. We define the ROC value as the time difference divide by the memory consumption difference.

**Definition 2.4** The ROC value of a join:

$$ROC = \frac{T_{\text{min}} - T(F)}{T(F)F - T_{\text{min}}F_{\text{min}}}.$$  

The ROC is a practical metrics to estimate the memory utility. A practical metrics should be simple enough and independent from the workloads. Formally, the marginal memory utility is the derivative of memory consumption. However, the calculation of marginal memory utility is a very complex function. In practice, the value $F$ is not continuous, the ROC is an alternative to the negative marginal memory utility in the discrete $F$. In Figure 2.6 c) and d) we show an example ROC curves of memory consumption curve in a) and b).

## 2.5 Memory Management based on Memory Utilization

In this subsection, we discuss the relationship between memory allocation and the performance of operator instances. The performance is directly affected by memory configuration on the ROC value of an instance of operators. After we examine the memory utilization of every operator, we have to discuss the aspects of competition among operators for main memory. Memory is mainly allocated for buffering cache and work area. Reading data from disk is relatively slower than accessing data from memory. When a file is read repeatedly, keeping data in memory after the first read can skip the IO cost for the following reads. The memory used for this propose is called buffering cache. Database operators need memory to execute its algorithm, e.g., a hash operator needs memory to build its hash tables. The memory used for this purpose is called work area. Memory used for both proposes are usually managed separately.

Memory used for buffer pool reduces ratio of data accesses that requires the disk reads. Memory used for work area reduces the disk usage caused by spilling the intermediate results. The separation of buffer pool and work area requires manual configurations to optimize the global performance, because the work area memory stores the intermediate results, which is going to be used several times for a short period. The memory tuning allows the system to maximize the buffer pool memory and meet the memory demand for work area on need. Therefore, the work area can be also used for buffer pool, when there is enough memory for all database operators in a period.

The memory management of buffering cache introduces replacement algorithms, e.g. LRU, LFU, Most Recently Used (MRU), and so on. The buffering memory is usually divided into multiple independent buffer pools. Each independent buffer pool is assigned to a specified object, such as indices, tables, hash tables, and so on. The memory tuning technique is used here to adjust the size of buffer pools. When the frequencies of some buffer pools exceeds others in a period, memory pages are shifted from one buffer pool to another. Therefore, the global hits of buffering cache are optimized.

The memory management of a work area depends on the type of database operators and the workloads. For a given operator, the ratio between the memory size and workloads determines the disk reads and writes caused by the operator. The memory tuning technique allows the database to allocate memory dynamically to all operators or adjust the memory size of their work area.
2.5.1 Memory Utilization of Operators

In Nephele, the records from an input may come from multiple channels. Nephele has three channel types: file channels, network channels and in memory channels. Using a file channel can exchange data via files stored in the local file system. Network channels are used for data transfer among different compute nodes and are implemented using TPC connections. In-memory channels are used for data transfer on the same compute node. To hide the complex channel implementations, Stratosphere uses the abstraction of streams for inputs and outputs. Therefore, the input stream can only be processed once. The size of input is unknown until the input is exhausted. For example, when some records of an input are from an output stream, we cannot know how many records are sent from the output stream until the output stream is exhausted. This characteristic of Stratosphere requires the operators to be designed in a flexible way to decide the best algorithm for a task by the memory allocation and the workload at runtime.

The IO reading and writing in Stratosphere is also implemented as a stream. If we want to spill data onto disk, the reading and writing access is fixed to sequential access. Random access to load pages from disk is not supported. We can traverse the stream by \texttt{next()} function to get the next record from stream or use \texttt{reset()} to set the read/write pointer to the beginning of the file. The cost of searching an single record is very high in Stratosphere, so the buffering of operators must consider the specific data reading and writing strategy to avoid unnecessary IOs. In the following part of this section, we will show the operators implemented in Stratosphere and their memory utilization.

**Nested Loop Join**

The widely used replacement algorithm is not a good choice for the IO of Stratosphere. The LRU is a suitable replacement algorithm for random accesses. However, in a scenario where only sequential file access is possible, LRU represents the worst case, because the LRU has no hits at all unless all data are buffered. The idea of LRU is to buffer the hot set, but by sequential access, there is no hot set. When we traverse a data file as a stream, all pages are traversed once. There is no hot set. In Figure 3.8 a) shows the LRU buffers a file. When we reset the stream, and traverse the file again the LRU evicts page 6 to load page 1, evicts page 7 to load page 2, ... No pages remain in the buffer while repeatedly reading the file. One exception is when then stream is reset frequently to make the first a few pages hotter than others.

A replacement algorithm for sequential access is MRU [NFS91, CD98]. The algorithm can buffer a part of data in memory, so when the buffered part of data is traversed there are hits. However, the MRU algorithm is not a good choice for Stratosphere, because that part of data in memory is floating, when MRU finished to read the buffered part, it has to seek the data file again to find the missing pages. In Stratosphere, the cost of seeking pages is the same as to read all buffered pages from disk. Figure 3.8 b) shows the situation of using MRU to buffer a file. When the stream is reset, page 1 and 2 can be read from the memory pages. However, when we access the page 3, the \texttt{next()} function has to be used to read records from the stream until it reaches the page 3, hence the MRU has actually no buffering effect by Stratosphere.

The memory allocation affects the performance of an operator in two ways. The proportion of the workload size to the memory size determines the best algorithm. The size of memory determines the IO costs that can be saved by buffering.

Both NLJ and BNLJ are implemented in the Cross. Both NLJ and BNLJ have two inputs \( R \) and \( S \). Figure 2.7 a) shows that the NLJ traverses the first input \( R \) for every record from \( S \). Using \( m \) pages with FPF replacement strategy, the IO cost of NLJ is \(|S| \times ((R) - m + 1), ((R) > m)\). When the inner dataset \( R \) fits into \( m \) pages, the IO cost is 0.

Using \( m \) pages, the BNLJ traverses the inner dataset \( R \) once for every \( m \) pages data from outer
2.5. MEMORY MANAGEMENT BASED ON MEMORY UTILIZATION

The additional memory can reduce the IO costs, only when the dataset $S$ can be built into fewer blocks. The IO cost of BNLJ is $\langle R \rangle \ast \langle S \rangle / m$. If $\langle R \rangle > m$, one more page for NLJ can save $|S|$ pages. For BNLJ one more pages saves IO cost $|R| \ast (\lceil |S| / m \rceil - \lceil |S| / (m + 1) \rceil)$ pages.

The output of NLJ is fixed, so the ordering of its output does not change. If either both datasets or only an outer dataset is sorted, the output of NLJ is also sorted. This characteristic allows replacing the BNLJ with NLJ. In the contrast, NLJ can not be replaced by BNLJ. The NLJ may have a better performance than BNLJ, when the arriving rate of $|R|$ is very slow, or when the $R$ is very small. Normally BNLJ has a smaller IO cost than NLJ, because it traverses the $R$ once for every block of $S$ rather than each record in $|S|$, but there is a special case: The NLJ has a higher performance than BNLJ, only when the $\langle R \rangle \leq M$. For both join algorithms, the size of the second input is unknown until the end of the algorithm. There is no difference to swap the inner and outer input. The BNLJ has to decide, which is inner before executing. Except when one page is reserved by the outer input, the rest is used for buffering the inner dataset. If the inner dataset fits into the $m - 1$ pages, the BNLJ should transfer to a NLJ to avoid any IO accesses. If the one page of the inner dataset is spilled, all pages should be spilled, and the best strategy is to use one page to read the inner and the rest for buffering the outer.

**Sort Merge Join**

The SMJ is implemented in Match. SMJ can have multiple sorters inside, the number depends on how many inputs are unsorted. For each unsorted input, it sorts the input first. When both datasets are sorted, the SMJ traverses them once to find the join partners.

The memory demand of SMJ greatly depends on the number of sorters. If there is no sorter created, the SMJ goes to the joining phase to traverse both datasets, so there is a very small memory demand to buffer the duplicated records from both datasets in joining phase. If one dataset is unsorted, one sorter is created, all memory allocated for sorters by the SMJ is assigned to the sorter. The running time of the sorter plus the time used for joining phase is the total time cost of the SMJ. If both datasets are unsorted, two sorters are created. Because we do not have any hints to the input datasets, we divide the memory for sorters evenly to both sorters. The faster sorter has to wait for the slower, so the executing time of the slower sort plus the time used...
CHAPTER 2. BASIC CONCEPTS AND RELATED WORK

for joining phase is the time cost of the SMJ.

Now we look into the sorter. Sorter is a memory tunable component; it must be implemented in another operator. In a sorter, memory is used as work area. The sorter performs a merge sort in two phases: sorting phase and merging phase. In the sorting phase, the sorter uses three threads to load, sort and spill the data. The memory is organized in memory blocks, which contain a bulk data and an index. To enable three threads to process the data at the same time, the allocated memory may be divided into multiple memory blocks, so each thread can get some to increase the parallelism. Memory blocks are used in a cycle through the three threads (the red route in Figure 2.8). The record is processed in the flow through three threads showed in the green route in Figure 2.8. Data records are read from the input channel into a memory block in the loading thread. After the memory block is full, the loading thread sends the block of data to the sorting thread. The sorting thread sorts the records by building the index. The sorted memory blocks are sent to the spilling thread. The spilling thread makes the decision whether the data in the block are spilled into a file channel or kept in memory. If the loading thread tells the spilling thread that the workload is larger than the memory allocation, the spilling thread begins to spill received data block to block. The spilled blocks are reused by the loading thread as empty blocks.

The proportion of the memory size and the workload is very important to the sorter. If the workload fits into one block, the sort algorithm actually chosen is quick sort. If the workload fits into the memory in multiple blocks, the sort algorithm is an in memory merge sort. If the workload exceeds the capacity of memory, the sorter goes to the merging phase and performs a $m$-way merge for the further sorting. By the $m$-way merge, $m$ blocks can be read at once. They are merged into one larger block and spilled again as one block. Meanwhile, $m$ blocks are erased and one merged block left. In each run, all blocks are read and written once, and the number of blocks are reduced to $1/m$. The SMJ begins the next run, unless there are less than $m$ blocks left. The IO cost is constant in the process. The buffering technique to one time sequential reading and writing has no effect. Therefore, when the whole data is spilled into disk, the sorter requires only one page for each opened data file.

The memory utilization of a sorter depends on the size of memory allocation and the size of blocks. The memory allocation determines whether there are merge runs with IO cost in merging phase. The size of blocks determines the number of spilled blocks and the IO costs. Given a memory allocation, bigger block means fewer memory blocks, but less runs and less IO costs.
2.5. MEMORY MANAGEMENT BASED ON MEMORY UTILIZATION

Figure 2.9: The IO cost of sorting with different number of runs

Figure 2.9 shows the blocks built by a 2-way merge. Both sorters have 20k records. The first sorter has a block to accommodate 4k records, and the second sorter has a block to accommodate 5k records. In the merge phase, the first sorter has three runs. The IO cost to merge all blocks for the first sorter is reading and writing 76k records. The second sorter has two runs, its IO cost is reading and writing 60k records.

As Figure 2.8 shows the sorter has three threads: loading, sorting, and spilling, the speed of threads cannot be the same. The memory blocks are reused in the cycle. The speed of the memory reuse cycle is bottlenecked by the slowest threads; therefore, more than three blocks are unreasonable. Given a workload in w pages, a memory allocation m, a corresponding number of blocks n ≤ 3, and the sorter can open b blocks at the same time, the block size is \( \frac{m}{n} \). For a fixed memory allocation, the sorter benefits from the multi-thread architecture by using more blocks, but it also costs more IO operations to merge spilled blocks. There should be a trade-off between the number of blocks and the size of a block.

If memory allocation is greater than the workloads, the time cost of the sorter is minimal. Otherwise, there are \( \lceil \frac{m}{mb} \rceil - 1 \) runs, and the IO cost is \( 2w\lceil \frac{m}{mb} \rceil \) pages.

In the joining phase, matched sets from both datasets are crossed by NLJ. In this process, several memory pages are used to buffer duplicated records from one dataset. It still causes some IO cost to spill the duplicated records, when their total size is greater than buffering memory. We set a proportion of total memory, which is used for the NLJ, and ignore the possible IO costs here.

**Hash Join**

Hash Join has two inputs. All memory in HJ is used for work area. In the building phase all memory is used to build the hash table. If the hash table, the records, and the structure of the hash table fit into memory, the IO cost is 0. Otherwise, the HJ has to partition the dataset \( R \) into smaller joins so that every smaller join fits into memory. The HJ creates a few partitions and assigns buckets into partitions (see Figure 2.10). Then the HJ scans \( R \), and uses the hash function to distribute records into buckets as well as the partitions. When the HJ is lacking of free memory, the HJ spills a partition into disk to free some memory to keep as many as possible partitions in memory. Define \( t \) to be the fraction of auxiliary structure of hash table in the memory. Given a workload \( |R| \) of building side dataset, the number of partitions \( p \) and memory allocations \( m \), the fraction of buckets in memory is in highest \( f = \frac{|m|}{(1 + t)|R|} * \frac{p}{p} \). The IO cost is:

\[
\begin{align*}
0 & \quad m > (1 + t)|R| \\
(1 + t)|R| * (1 - f)p & \quad m > |R|/p \\
(1 + t)|R| & \quad \text{else.}
\end{align*}
\]
In the probing phase, HJ begins to scan the dataset $S$. For each record $s$ from $S$, HJ uses the same hash function on $s$ to find its bucket. If the bucket is in memory, HJ evaluates the join results for the $s$ and its join partners. Otherwise, the bucket is spilled; $s$ is also spilled into the probing side file channel associated to the bucket’s partition. The IO cost is estimated as $|S| \times (1 - f)$. For each spilled partition in disk, HJ tries to load the whole partition into memory. If there is enough memory to rebuild the hash buckets and records of building side dataset in the partition, HJ loads this partition into memory. HJ hashes each record from the probing side channel to find the join partners via the hash table. If there is not enough memory, HJ partitions the spilled partition again to split the partitions into smaller partitions recursively until every partition fits into memory. If the partition fits into memory, the IO cost to load the hash table is estimated as $(1 + t)|R| \times (1 - f)p$. The average size of each bucket is $(1 + f)|R|/p$. If the partition has to be partitioned recursively, the IO cost is analog to the cost in the building phase. After the dataset $S$ is scanned, the maximal memory demand of memory is to load the biggest partition. The rest memory can be released.

**Reduce, Combine and DataSink**

The Reduce implements a sorter. The memory utilization is similar to the sorter introduced in SMJ. The IO cost of Reduce is smaller than a sorter. The Reduce and Combine aggregate the data, so the dataset is smaller after blocks are merged. Figure 2.11 shows an example of word count. Records with the same key are aggregated to one record. The actual IO depends on the aggregating UDF and the workload. DataSink also provide a sorter to give the result of a join an ordering. The performance is exactly like that a sorter.

**2.5.2 Memory Competing between Operators**

The ROC model is used to measure the memory utilization of operators. Our memory tuning approach takes the concept of class fencing [BCL93]. There is a memory consumer, the Combine operator that does not take part in the memory tuning. The Combine operator implements an in-memory reduce operator. It is possible to reconfigure the memory allocation of a Combine operator,
but its memory utilization is theoretically constant. However, the memory size determines how many records take part in the Reduce. If the memory size is too small, it is possible that no records are combined. Therefore, we define the Combine as a non-tuning memory consumer, the user decides a fixed memory configuration to the operator, otherwise the Combine will accumulate all available memory pages or give up all memory pages for its stagnating ROC value.

Although the memory used for NLJ is normally classified in buffer pool memory. Because operators never share the memory in the buffer pool, we consider the memory used for NLJ as work area. Therefore, the memory tuning mechanism is occurred in work areas. All memory pages used for buffer pool are available for replacement, unless a page is locked by an operator. The system can estimate the performance of all buffer pools and give a solution for all buffer pools. The reconfiguration can be executed immediately without any constrains. It is difficult to estimate the performance of operators for work area memory, and the function of marginal memory utilizations are varying from each other. It is impossible to find an optimal solution with linear programming. The optimal solution might not be a feasible solution for all operators too. The memory management in side of operators is also different from each other. For example, the memory used for building hash tables has its own organization. Only adding a special number of pages can improve the performance, otherwise the additional memory pages are waste. Another problem is that some operators have a relatively high ROC, but do not require any additional memory. For example, when a sorter has gone to merge phase, the algorithm is fixed, more memory does not help. On the other hand, an operator can have a relatively low ROC, but it cannot give up any memory pages. For example, a SMJ has gone to joining phase, there is no memory available to release. The time point of insert memory pages into an operator and require memory pages from an operator is essential. For example, the memory pages in BNLJ are buffering records, which have no copies in the disk. It can release memory when all buffered records are joined with the inner dataset. If the system obliges the BNLJ release memory for memory tuning, the BNLJ has to spill some records into disk and the join ordering is in a mass, therefore, the algorithm of BNLJ is disturbed by the memory tuning. In this case, the benefit of the memory tuning might be counterbalanced by its negative affection to the BNLJ. If an operator can accumulate enough memory to run in cache, it can escape from the memory tuning to prevent additional cost by changing the algorithm. Those operators running in cache are normally serving a small workload or there is enough free memory in the system, we let them run until the end.

As a summary, the obligations on operators do not work; further, more obligations tend to hinder the improvement of overall performance. Therefore, a global solution has no effect, because the system cannot oblige an operator to release memory or receive additional memory. The memory competing should consider the local situation of operators. Interim findings can be summerized as follows:

- An operator has the right to refuse the memory reconfiguration to keep its algorithm.
- An operator has the right to decide how many memory pages it wanted to fit its algorithm.
• An operator has the right to decide when to take part in the memory tuning without disturbing its execution.

• An operator running in cache is excluded from the memory tuning to prevent the additional cost of algorithm changing.
Chapter 3
Memory Tuning for Stratosphere

3.1 Problem Formulation

3.1.1 Goals of Memory Tuning

In this thesis, we implement a memory tuning conception in Stratosphere using a feedback technique. There are six memory consuming operators, namely NLJ, SMJ, HJ, Combine, DataSink and Reduce. The memory tuning components should reconfigure the memory allocation of them at runtime to achieve a higher global performance of each computing node in the Stratosphere system. For the simplicity, if we say operator, we refers to an instance of an operator.

The memory tuning components gather the feedback information from running operators. The performance of the operators is estimated by both feedback information and historical statistics. The memory utilization is measured in ROC value. The operator with a higher ROC value has the right to get more memory, in addition the loser should give up some to the winners.

The memory tuning components should be chosen with a light weight approach. Stratosphere is a high performance big data analytic platform. The cost of memory tuning should be small enough, so that the memory tuning will not improve the performance. The memory tuning components should not disturb the execution of the operators and should not change their basic algorithm.

3.1.2 System Architecture of Memory Tuning

The memory tuning component is implemented on the physical level of Stratosphere and the memory tuning is transparent to users and programmers. Before we go deep to the physical level of the Stratosphere, we use an query as an example to explain the Stratosphere on the logic level. After that, we describe how Stratosphere map the query from logic level to the physical level structures. Finally, we present the system architecture of the memory tuning component, and reveal how it works with the query on the physical level.

We start with a SQL query (see Appendix B.3) as an example, to show how memory tuning is taking effect. A query in script is translated to a logic structure in the PACT program (Figure 3.1). The PACT program consists od four operators:

- Map operator to screen out the records which do not match the date predication,
- Match operator to join the two tables *lineitem* and *part*,
- Reduce operator to aggregate revenue data,
- Datasink operator to emit the result.
CHAPTER 3. MEMORY TUNING FOR STRATOSPHERE

The MAP filters the input channel without using memory for buffer pool and work area, so it is not a memory consuming operator. The memory consuming operators in the example are: Match, Reduce and DataSink (see green boxes in Figure 3.1).

![Figure 3.1: PACT Program of TPC-H Query 14](image)

The architecture is shown in Figure 3.3. The memory tuning operator has a local memory manager to perform the memory tuning functions. The local memory manager holds a configuration to denote the memory allocation status. An operator may have memory consuming components. For example, a sorter is a memory consuming component. A sorter alone meaningless; it has to be implemented in an operator for a special purpose. In our example (see Figure 3.1) the Match operator drives two sorters to sort both input streams, performs a merge join on both sorted streams, and then runs a matcher to emit matched records (see dashed box in Figure 3.1). The operator and its components have to allocate, release and tune via the local memory manager installed in the compute nodes. The operator and components can manage allocated memory assigned by the local memory manager. There is a performance estimation component to measure the memory usages of the operator. The performance of a component is considered as a part of the operator, it is the operator’s duty to optimize the memory allocation for itself and its components. The result of the performance estimation is used to decide whether the operator should get more or give up some memory pages. The operator has a longer life cycle than its components. When the operator is aborted or canceled, its components are also aborted and canceled. In this case, the component is asked to release all memory pages and other resources.

The PACT program is compiled by the PACT Compiler into a Nephele JobGraph, which is in turn translated into a Nephele execution graph. The corresponding execution graph for Example 3.1 is shown in Figure 3.2. Here we choose the degree of parallelism of four for the Map and Match operator, Reduce and DataSink only have a degree of parallelism of 1. Two compute nodes in the system are responsible for the query. Stratosphere splits the input lineitem into four sub-channels and sent them to four instances of Map operator. The results of each instance of Map operator
3.1. PROBLEM FORMULATION

![Diagram of Nephele DAG of TPC-H Query 14](image)

Figure 3.2: Nephele DAG of TPC-H Query 14

The input `part` is broadcasted to every instance of Match operator to perform a SMJ. The results from Match are sent to one instance of the Reduce operator to aggregate the final result.

### 3.1.3 Problem definition

The problem to solve is to adjust the memory allocation of all memory consuming operators at runtime to improve the overall performance in every compute node.

The memory is divided into page frames with fixed identical size, and there are $P$ pages. All memory pages used for buffering and working area such as sorting and hashing are managed by a memory manager. Memory tuning gives the memory manager a solution to optimize the memory allocation of all memory consuming operators.

We have a set of instances of memory consuming operator $O\{o_1, o_2, \ldots, o_m\}$. The instances of an operator is the object physically keeping memory pages and holding instances of memory tuning components. The memory tuning is actually performed on the instances. Every memory consuming vertex in the Nephele DAG is an instance of the memory tuning operator.

We have sets of Input $I\{I_1, I_2, \ldots, I_l\}$ In the system every input is a channel. $I_m$ is the set of inputs of an instance operator $o_m$. If $o_m$ has components, all inputs of its components belong to $o_m$. It is possible that some channels share the same resource. In this case, Stratosphere copies or splits the input channel according to the logic of the PACT program. It is transparent to the operators, so we consider them as different from each other.

Each instance of operator has a memory configuration, therefore we also have a set of configurations $C\{c_1, c_2, \ldots, c_n\}$. A configuration contains the information about the memory allocation, additional memory demand and the memory request from the memory manager of the instance of operator, and its components. The additional memory demand and memory request of an operator are managed by the memory tuning algorithm. The additional memory demand and the
memory request of an instance of the memory tuning operator is set by the holding operator by their performance.

The performance of an instance of operator is $T_n = \text{Time}(I_n, c_n)$. During program deployment Stratosphere creates the operator instances of an execution graph. However, until the arrival of actual input an operator instance is idle or waiting. Thus, we measure the execution time of an operator rather than its overall lifetime in a query. The timestamp $T_n$ denotes the arrival time of the first input item. Here we consider only from point when the first record arrived up to the last memory page is released.

The overall performance of the system is maximized when the execution time of all instances of operators is minimized. So we get

$$
\text{Min} \sum_{i=1}^{n} \text{Time}(I_i, c_i)
$$

subject to:

$$
\sum_{i=1}^{n} c_i.\text{allocation} \leq P
$$

The performance of an operator is measured in ROC value. The ROC value of the operator $n$ is: $\text{ROC}_n = \text{ROC}(I_n, c_n)$. When an operator has more than one memory tuning component, it has to aggregate the ROC value by querying the in execution time from the component.

When $\text{ROC}_p > \text{ROC}_q (0 < p, q \leq n, p \neq q)$, the overall performance is always improved by shifting pages from $o_p$ to $o_q$.

### 3.2 ROC Estimation

In this section, we discuss the ROC estimation. The ROC depends on the memory allocation and the running time. The memory allocation is very clear, but the running time is unknown.
3.2. ROC ESTIMATION

until the operator is finished. The ROC estimation is therefore converted to the running time estimation. Unfortunately, in Stratosphere the input dataset is also unknown until the input datasets are fully loaded. When an operator wants to allocate more memory, the only thing known is that the workload exceeds the memory allocation. To tackle this problem, we make an optimistic assumption, that the workload fit into the memory an operator currently asked for, if the workload is partially unknown. When the workload is known better at runtime, the running time estimation is more precise. There are many kinds of time cost in an operator.

- The cost to load records from the input stream is \( C_{\text{load}} \). The resource of the input stream can be file, network, or in-memory. That depends on the dataflow of a query. The arriving rate of records is essential to \( C_{\text{load}} \). When the input stream has a mixed resource, the arriving rate is unstable.

- The cost to deserialize the record is \( C_{\text{deserialize}} \). Recall that the records from a channel are in a binary array. The operator has to scan the binary array to find out where a record starts and ends. A new structure of the record is rebuilt and the values are assigned.

- \( C_{\text{org}} \) is involved, when the operator moves a record from one buffer to another or inserts it into a data structure. For example a record is inserted into a block for sorting, an entry is created for the index. After a record is hashed, the record is inserted into the hash table.

- \( C_{\text{hash}} \) is used to compute the hash code of a record. With an appropriate hash function the cost of computing the hash code of record is constant. However, for every record in both relations the hash function is invoked once.

- \( C_{\text{comp}} \) is the time cost to compare two records. The compare function is called in many situations. For example, the sorting uses the compare function intensively to ordering the records.

- \( C_{\text{pred}} \) is used to check, whether two records satisfy the filter predicated.

- \( C_{\text{output}} \) is the cost to emit a result record into the output channel.

Although it is known an operator spends its running time, a clock cannot be set to measure it all the time, the cost is too high. The time cost is divided into two parts: Fixed time cost \( C_{\text{fix}} \) and flexible time cost \( C_{\text{flex}} \). The total cost of an operator is \( C_{\text{total}} = C_{\text{var}} + C_{\text{fix}} \). The estimated total cost is

\[
C_{\text{total-est}} = C_{\text{var-est}} + C_{\text{fix-est}}
\]

Given a workload, unlimited memory allocation, and other parameter is unchanged, the time cost of a specified operator is theoretically the same for every execution. We consider this value as \( C_{\text{fix}} \). When the memory allocation changes, the operator has IO cost and sometimes other costs, e.g., the Hash Join may have a recursive partitioning. Expect the IO cost and other expected time, e.g., the recursive partitioning of HJ, the rest of time cost is caused by the complexity of the operator. The ROC estimation is taking place in runtime, the estimation can be done by measuring the time cost and by using the historical statistics. Each operator starts with a memory allocation. Until it takes part in the first memory tuning, it can keep the memory allocation or release them on its own. Before the operator gives the first ROC estimation, it can serve the workload for a while. The time cost and the IO cost can be measured. \( C_{\text{fix}} \) until this point can be calculated as \( C_{\text{fix-now}} = C_{\text{measured-total}} - C_{\text{IO}} \). If some operator has more than one phases, we measure the time cost for each phase. At the beginning some phases have not started yet, for this part of time cost, the time cost from the historical statistics is consulted, until real observation are available.
Here a simple solution to estimate the total fixed costs is applied. If the complexity of operator \( o \) is \( O(M \log M) \), whereby \( M \) is the size of workload, there exists \( a \) and \( b \) satisfy

\[
aM \log M \leq T(o) \leq bM \log M
\]

Here \( a \) is the parameter taken by the fastest primitive operation and \( b \) is the parameter taken by the slowest primitive operation. If an operator has a complexity \( O(g(M)) \), the fixed cost is defined constant as

\[
C_{\text{fix-constant}} = C_{\text{fix}}/g(M)
\]

The fixed cost constant is used in the worst case to estimate the fixed cost of an operator:

\[
C_{\text{fix-est}} = \max(C_{\text{fix-constant}}) \cdot g(M)
\]

The operator has to spill one page first and then pick up a free page, except when it picks up free memory. Because the IO reading and writing is take over by IO Manager, the page put to spill and the page picked up can be different, we measure the time gaps between the operator sends a page for spilling and the time it gets one freed pages as the IO costs. The average time IO cost depends on the strategy of the operator, so we consider the IO cost is different between them. By measuring the average time and the total IO cost, we can estimate the \( C_{\text{IO}} \). By additional work caused by memory allocation, e.g., the recursive partitioning of a Hash Join, we can simulate the recursive partitioning, and estimate the additional cost by the observed fixed cost constant and the additional IO cost.

### 3.3 Asynchronous vs. Synchronized Memory Tuning

In the previous section we have described the problem arisen by interrupting operators at runtime (Figure 3.4 a). As a result it was detected that there should be no operators that are forced to lose or get memory pages. Operators are granted a high autonomous right to their memory allocation. Given these findings, we now present two approaches for memory tuning; synchronous and asynchronous memory tuning. We compare these two memory tuning methods, and show that the asynchronous memory tuning is superior in performance for the work area memory tuning.

Synchronized memory tuning (SMT) is a mechanism to ask an operator to shift a certain number of pages to another operator. Figure 3.4 shows a workflow of the SMT. An operator is ready for the memory tuning, when it has finished to process data in the current work area memory and the data needed has been backed up somewhere. We have to wait until both operators are ready. Then, they can negotiate the amount of memory transfer from the loser to the winner. After the memory is transferred, both of them continue their execution. SMT is a feasible method of memory tuning, but it has many disadvantages. For instance, two operators cannot be in appropriate status at the same time, so there is a synchronization between them, that means one operator must be suspended and wait for the other. Suspending an operator negatively affects the performance of an operator. Suspending and investing more memory to a operator are two conflicting options. For the winner operator, which has invested too much memory to get too little performance, the suspending decreases its ROC. It is possible that after the suspending it becomes the loser operator. The SMT is easy to implement, but the flaws are also obvious. SMT is common in memory tuning for buffer pools. The buffer pools are highly adaptively to memory changing, the time cost is shifting unlocked pages from one pool to another. However, work area memory contains intermediate data, which has no copies. Shifting work area has to wait until the operator is stopped to process the data. Therefore, the waiting time is longer than the buffer pool memory.
3.3. ASYNCHRONOUS VS. SYNCHRONIZED MEMORY TUNING

Figure 3.4: Workflows of different memory tuning techniques.

a) memory tuning with interruption

b) Memory tuning with synchronization

c) Asynchronous memory tuning
Now, we introduce an asynchronous memory tuning (AMT) to tackle this problem in Figure 3.4c). We show a case in TPC-H Query14 to demonstrate this procedure (see Figure 3.5). Assuming, that there are to match operators of Query 14 in the same compute node, Match 2 is the winner and now it is ready for memory tuning, it routine is marked in red. It firstly finds out how many memory pages it desires and the minimal memory it can accept. In the second step, it computes its ROC value and sends to the memory manager. In Step3, the memory manager consult the ROC table. This time Match 2 is the winner. If there is enough memory to fulfill at least the minimal memory requirement, the memory manager allocates more memory for Match 2. If there not enough, the memory manager records the demand from Match 2. Whatever the result of the memory tuning is (allocate more, release some pages or nothing changes), Match 2 reconfigures its memory setting and continue to work.

Match 1 is a loser operator; its routine is marked in blue. It does same thing to the Match 2 in Step 1 and 2. It sends the memory requirement and its ROC to the memory manager. The memory manager consults the system and finds out Match 1 is a loser in Step 3. For the loser operator the manager has to check the reserved memory amount to decide whether the loser operator can get more memory in Step 4. If all memory is reserved, it consults the ROC table again to find out whether there is an operator in a winner group still in lack of memory. If so, the loser has to release memory to the memory manager, otherwise, the loser operator can skip the releasing. In this case, memory manager finds out that the requirement of Match 2 has not been fulfilled yet. Match1 is asked to release memory for Match 2. Match 1 gets the memory tuning result; it reconfigures its memory setting and releases memory to Match2. Next time when Match1 is ready for tuning, it can take the free memory directly from the memory manager.

Under AMT mechanism, no operator is asked to wait for another, but the memory pages wait
for operators. However, AMT also has a disadvantage. It is possible that an operator transfers memory for nothing, because the winner operator might not require any additional memory when the loser is ready. The disadvantage of AMT can be corrected by allowing multiple operators to take part in one memory tuning operation.

3.4 Memory Tuning Algorithms

In the previous section, the memory utilization of operators and the interaction between them has been discussed. In this section we show the details of our approach to memory tuning for the work areas. Inspired by the class fencing [BCL93], we introduce an asynchronous memory tuning to tackle the problems shown in the previous section.

3.4.1 Memory Competition Algorithm Based on Memory Utilization

The AMT has the disadvantage that the memory provided by the losing operator may not be actually used by the specified winner operator. To tackle this problem it is best to use the central memory manager as a broker rather than as an entity specifying the origin and the destination of the memory transfer. We divide all managed operators into winner and loser groups according to their ROC value. Figure 3.6 shows an example of the ROC table. For a winner operator, all operators in the loser group are candidates for memory tuning, conversely, for a loser operator the whole winner group is its memory tuning partner. The memory pages provided by an operator from the loser group from an operator in the loser group are merged into the free memory in the central memory manager. When a winner operator requires memory, it consults the central memory manager whether its requirement is affordable. If the requirement is affordable the winner operator can get the memory pages at once. For example, operator 1 requires 10 pages, if there are 10 free pages in the memory manager, operator 1 can get the free pages rather than waiting for the loser operators to release 10 pages. That means the operator do not have to wait for a loser operator releasing memory for the specified operator’s requirement.

Memory utilization competing between two groups has many advantages to the competing between two operators. The competing is successful, only when the winner requires an affordable
memory transfer from the loser. It is possible that multiple loser operators should downsize their memory configuration for a winner. Using the central memory manager as a broker makes it possible to ask several operators to put downsized memory pages together and wait for the winner operator. The memory tuning is toward the running operators. New operators may be blocked by running operators when large workloads are running in the system. This situation is mentioned in the 'Managing Memory to Meet Multiclass Workload Response Time Goals' [BCL93] in the case of large class with available memory policy. Putting the freed memory to the central memory manager prevents a large class eating up all memory.

When the system has considerable free memory, it is also possible to assign more memory pages for loser operators. For example, an operator released many memory pages after it is finished. Here, we calculate a boundary of reserved memory amount for winner operators. For example in Figure 3.6, the winner operator asks for 14 pages. When the free memory amount exceeds this boundary, e.g., there are 100 pages free. 86 pages are laid-off. At this moment, no more free memory is needed for winners, so the loser operator is also allowed to get additional memory to improve its performance. The memory tuning is therefore automatically disarmed, because it is unnecessary to compete for resources. As shown in the example, we record all unfulfilled requirements of operators to reflect the overall memory shortage. The ROC table is dynamically updated, when the memory setting of an operator is reconfigured or an operator is finished, the ROC table changes as well as the reserved memory amount.

### Algorithm 3.1 Algorithm of Memory Tuning

```
1: function MemoryTuning(operator, demand, minial)  
2:     if operator is winner then                   ▷ winner
3:         ROC = getROC(operator)                  
4:         update(operator, ROC)                   
5:         if freePage > minimal then              ▷ too many free pages
6:             allocate(operator, MIN(demand, freePage))
7:             return                               ▷ record unsatisfied requirement
8:     end if
9:     update(operator, demand)                    ▷ record unsatisfied requirement
10:    return                                     ▷ loser
11: else                                                                 
12:     available = freePages − sum of unsatisified requiremet of winner operators
13:     ROC = getROC(operator)                     
14:     update(operator, ROC)                     ▷ too many free pages
15:     if available > minimal then
16:         allocate(operator, MIN(demand, available))
17:         return                               ▷ record unsatisfied requirement
18:     end if
19:     update(operator, demand)                  ▷ record unsatisfied requirement
20:     request the unsatisfied requirement of the highest winner operator
21:     return                                    
22: end if
23: end function                                 
```

The interaction between memory managers is a handshake negotiation. The central memory manager keeps the ROC values and memory requirement of operators. Figure 3.1 shows the pseudo code of the algorithm, which runs in operator and memory manager. According to the ROC table, half of the operators with higher ROC are winner operators, the rest are loser operators.

When an operator is ready for memory tuning, it reports its memory requirement, both the actual demand and the minimal acceptable amount (Step 1).
When an operator asks for more memory, the memory manager checks its ROC value to decide which group the operator belongs to (Step 2 line 3, 13). If the operator is a winner operator, it is allowed to get free memory according to its requirement (Step 3 red, line 2 - 10). The memory inserts memory pages to the operator when the minimal acceptable amount is affordable. If the operator is a loser operator, the memory manager checks the reserved memory amount and the available free memory to decide whether the memory tuning should be turned off. If the memory tuning is off, the loser is also allowed to allocate more memory pages (Step 4 green, line 15 - 18). If the memory tuning is on, the memory manager finds an unfulfilled requirement of a winner operator and sends this requirement to the operator to ask memory release (line 20).

The memory requirement is updated. Although the memory requirement of a loser operator (line 19) will not accept at this time, a loser operator might still become a winner in the future, because some operators might be completed and a winner operator also might become a loser when it gets additional memory. The new ROC value updates the grouping of operators and affects the following memory tuning operators.

3.4.2 Summary

In this chapter, the asynchronous memory tuning was introduced to allow memory tuning without disturbing the executing of operators. Based on the asynchronous memory tuning, memory tuning has been extended between two operators to two groups. Transferring memory from an operator from the loser group to an operator from the winner group improves the overall performance.

The memory manager is used as a broker of memory tuning. All operators connect to the memory manager, there is no interplay between operators. The downsized memory by memory tuning is considered as free memory. All free memory is reserved for new operators and winner operators. When there are too many free memory pages, there is also a mechanism to disarm the memory to allow all operators to allocate enough memory, as they want.

In this chapter, it was also presented how the ROC value is estimated by the running information and the history statistics.

3.5 Memory Adaptive Joins

To realize the memory tuning, the memory tuning component has to be designed in the memory manager and the operators also have to be adaptive to new reconfigurations for both memory downsizing and upsizing. The changed memory allocation should be reflected in the performance of operators. If an operator has the same performance with additional memory pages, there is no reason to assign more memory resource to it. The flexibility of memory allocation is essential to memory tuning. If it is very costly to insert or release memory pages for a new configuration, the cost of memory tuning eclipses all benefit brought by memory tuning itself. In this section, it is discussed how to make operators adaptive to new memory configurations.

3.5.1 Workload Discovery

Workload discovery is introduced to tackle the memory shortage problem [DZ02]. There the prediction of memory demand cannot be very precise. There are also memory fragments, which cause that the memory allocation cannot be used perfectly. The size of intermediate results and the auxiliary structures is also bigger than expected. In this case, workload discovery allows an operator to make up for the memory shortage by imprecise memory predictions.

In Stratosphere, the size of the input is unknown; this concept is also implemented in the memory tuning. Initially the operator can start by a small amount of memory. It discovers the
actual size of inputs by loading the input. When it finds that the input size becomes larger than the current memory size, it tries to allocate more memory to increase the capacity by a fixed factor (e.g., 10%) or a given size (e.g., 4 MB).

The workload discovery is a memory tuning mechanism to improve the performance. There is a difference between the workload discovery and other memory tuning methods. Because the workload discovery happens when the input is loading, no data in the memory has a copy on the disk. To prevent unnecessary IO costs, an operator is allowed to use workload discovery until it has to spill its data onto disk. Once the operator has spilled data into disk, it is impossible to run in memory anymore, so the workload discovery period is over. If the operator wants additional memory, it has to use the memory tuning mechanism.

### 3.5.2 Cycles for Adapting Memory

The design of all memory adaptive operators is in cycles (see Figure 3.7). If the data cannot fit into the memory, the operator has to reuse the memory pages. Before the operator reuses them, all data in the memory pages is useless, the operator is ready for memory tuning. More memory pages can be inserted or some pages can be released. After memory tuning, the structure is rebuilt with new memory allocation for the following execution. Sometimes the settings and algorithm are also reconfigured after the memory tuning to adapt the operator to the new memory allocation. The operator runs in the execution, memory tuning, and reconfigure cycle repeatedly, until all data are processed. The life time of this cycle is the time cost considered in this study.

### 3.5.3 Memory Adaptive Nested Loop Join and Block Nested Loop Join

Here we introduce a special replacement algorithm for resettable stream: Front Page First (FPF). The idea is similar to MRU, using the memory to buffer a part of data in memory to make hits when the stream is traversed again. The difference is that it keep pages from the end of a file in the buffer all the time. When the current page of the stream hits, all allowing pages are buffered. When the FPF reads all records from the buffer, it resets the stream to read from the head, seeking is not appeared. A queue is used to store the pages and a pointer P to show the boundary of the buffered position. At the beginning \( P = 0 \). When the stream is traversed for the first time, the first page is always evicted in the queue to load the next page and insert the next page at the end. When the first time an End of File signal (EOF) from the stream apears, \( P \) is set to the second page in the queue (see Figure 3.8 c). The first page is left for pending. When the stream is reset,
3.5. MEMORY ADAPTIVE JOINS

only the first page is used to load new pages and insert the page in front of P. When page 1 is accessed, page 6 is evicted. When page 2 is accessed, the page 1 is evicted. After page 3 is accessed, the situation in Figure 3.8 d) occurs. When the stream is reset, or gets an EOF, the position of P is updated. If n more pages are inserted into the FPF, n + 1 pages are used to load missing pages. Figure 3.8 e) shows the situation that the FPF uses an additional page to load page 4 from the situation d). After traversing the file 3.8 f) occurs. N new pages are merged to the loaded part of the data file when P is updated. If the FPF is required to give up n pages, the FPF gives up the first n pages in the front of queue and update the position of P. Figure 3.8 g) shows releasing 2 pages from d). When continuing to access page 5, the page 7 is evicted, so we get the Figure 3.8 h) occurs. Therefore, FPF increases the flexibility of the memory size. It is a helpful character to the conception of memory tuning. In the Table 3.1, the performance of FPF is compared to LRU and MRU when loading m pages data sequentially with n pages for buffering.

The NLJ and BNLJ are based on repeating sequential reading on both datasets. MRU is not suitable for buffering a stream, but the FPF replacement algorithm copes to streams and it has a similar performance like MRU.

We have two Nest-Loop-Join algorithms, NLJ and BNLJ. All memory pages of NLJ Algorithm A.1 is managed by FPF replacement to buffer the first relation. The FPF replacement algorithm

\begin{table}
\centering
\begin{tabular}{|c|c|c|c|}
\hline
 & LRU & MRU & FPF \\
\hline
IO cost for loading & M & M & M \\
IO cost for reading(n < M) & M & M - n & M - n + 1 \\
IO cost for reading(n = M) & 0 & 0 & 0 \\
\hline
\end{tabular}
\caption{Performance comparison of replacement algorithm}
\end{table}
inherits the high flexibility from buffering replacement management, when all records have a copy on disk. We are scanning the relation R and insert it into the buffer with FPF replacement (line 3-11). If the buffer has not spilled yet, we use workload discovery to solve the memory shortage (line 1-8). NLJ is always ready for memory tuning, if the whole relation R is spilled. So after buffering R we check whether the buffer is still in memory. If the buffer is once spilled, we spill all data (line 12-14). While NLJ is scanning the relation S, we do not like to perform the memory tuning too often. We set the memory tuning interval, e.g., to perform memory tuning once after traversing the inner relation R 100 times (line 18). By memory tuning (Algorithm A.4 line 1-10), NLJ can get more memory (Algorithm A.4 line 9) or asked to release some pages (Algorithm A.4 line 7).

The BNLJ (see Figure A.2 has two models. Similar to NLJ, the BNLJ uses the same strategy to load the first relation with workload discovery (line 3-17). If the first relation fits into the memory, the BNLJ is transformed to a NLJ, and there is no more memory tuning (line 17, 37). In this case, the BNLJ can execute without IO cost until the end. If first relation is spilled, the complete first relation is spilled into disk. Otherwise, all memory pages are used to buffer the dataset S (line 21). From now on, the size of the first relation is known, so the size of the first dataset can be set as the maximal memory demand of the BNLJ. Once the BNLJ reaches the maximal memory, the BNLJ is transformed to an in memory NLJ. BNLJ begins to load the dataset S. Every time when the buffer is full, a memory tuning point is inserted trying to enlarge the size of buffer for S (line 25). If the memory manage grants the BNLJ more memory space, it continue loading the S. If the memory manage requires the BNLJ to release memory, it begins to join the records in the buffer with the R and then releases memory after the joining (line 27 - 35). After the releasing, it begins to load more records from the second relation again.

3.5.4 Memory Adaptive Hash Join

In the building phase the memory demand of HJ (Algorithm A.6) is relatively bigger than in the probing phase. Workload discovery is used in the building phase to accumulate more memory pages (line 5). If a partition is spilled, the workload discovery is off. Before one partition is forced to spill, a memory tuning is performed to try to solve the memory shortage problem (line 7). If the HJ is asked to release memory, it spills partitions to free memory and release them by memory tuning. If the memory manager asks the HJ to release memory, partitions are spilled to get more free memory (line 13).

In the probing phase, the probing side dataset S is scanned. If the record from S, say s, it hits a partition p in memory, HJ finds the join partner to evaluate the join result (line 15 - 17). Otherwise, s is written into the probing side stream of P (line 20). The partitions in memory are deleted after S is scanned (line 23).

After the probing side input is scanned, the maximal memory demand is changed to the size of the maximal partition. If memory inside is bigger than the maximal partition, unnecessary memory is released (line 26). If memory inside is smaller than the maximal partition, we use memory tuning mechanism is used to try to reach the maximal memory demand to avoid recursive partitioning (line 28). The memory tuning here ignores the memory requirement from the memory manager to prevent the high cost of recursive partitioning (line 30). For simplicity, the recursive partitioning in the algorithm is skipped. The spilled partition is loaded into memory (line 28), and the records from S which are hidden in the partition are probed again (line 30 - 31).

3.5.5 Memory Adaptive Sort Merge Join and CoGroup

SMJ has two sorters, the memory used for the SMJ is to cross the duplicated records, we ignore them. We focus on the memory tuning of the sorter in Algorithm A.7.
3.6. SUMMARY

The loading thread (line 2 - 15) loads records from the input. When the loading thread finds there is less than one block in the free memory block array, it sends a signal to the spill thread (line 23 - 62). When the loading thread has loaded a block fully, it sends the loaded block to the sorting thread (line 17 - 21). If the input is exhausted, the loading thread sends the EOF to the spilling thread. When the sort thread gets a loaded block, it sorts records in the block and sends it to the spilling thread.

The spilling thread checks the spill signal from the loading thread. When it gets the spill signal, it tries to get more memory to solve the memory shortage problem by workload discovery and memory tuning (line 28). If the memory manager asks the sorter to release some memory or no more memory is allocated, the spill thread chooses some sorted blocks, merges them in memory, and spills the merged block onto disk to free some pages (line 45). Free pages can be used to release for memory tuning (line 47). If more memory is allocated it builds new blocks for load thread (line 43). When the spilling thread gets sorted blocks from a sort thread, it keeps them in memory (line 54). After the spilling thread gets the EOF, it merges all blocks in disk with m-way merging algorithm (line 59). The sorted blocks in memory are also merged by in-memory 2-way merging algorithm (line 60).

To maximize the effect of memory tuning to the performance of the sorter, the sorter is allowed to keep blocks in memory to reduce the spilled part of data. The blocks in memory are merged by in-memory merge separately. In the merge phase, the sorter does not need any additional memory, but memory tuning points are still inserted into the sorter between two merge procedures to allow the sort to release memory for better usage. The sorter picks up several blocks in memory and merges them with blocks in disk together by m-way merge. This merging frees blocks and provides more free memory pages.

When a sorter has a relatively small input, it may allocate more memory than the workload. The sorter can transfer the memory to the SMJ. When the memory manager asks the SMJ to release memory, it releases the free memory first. The free memory is also available for another sorter.

In the join phase of SMJ, no memory tuning is possible.

3.5.6 Memory Adaptive Reduce and DataSink

The Reduce implements a sorter with a reduced UDF. The sort implemented has little difference to the SMJ. The merge procedure of Reduce is an aggregating function. The in memory merge is impossible, because the aggregating function modifies the data. If the workload does not fit into memory, an m-way merge procedure is used to merge smaller blocks into disk rather than keep them in memory in merge phase.

The DataSink implements a sort to give an order to the results. The sorter used is exactly the same to as used in the SMJ.

3.6 Summary

In this chapter, main memory tuning components for Stratosphere is discussed. The ROC values of operators is used as a metrics for memory utilization competition. The problem definition is in the section 3.1. The ROC estimation method is described in the section 3.2. The asynchronous memory tuning algorithm is introduced in the section 3.3 as well as the corresponding memory sharing mechanism between operators in the section section 3.4. Finally, the design of memory adaptive operators to cope with the memory tuning memory manager is presented in the section 3.5.
Chapter 4

Evaluation

4.1 Implementation

We integrated a prototype for memory tuning in the Stratosphere system using the Java programming language version 1.6. The basic design of the memory tuning manager comprises two parts, namely a basic memory management component, and a memory tuning component.

Figure 4.1 shows the original non-memory tuning approach of Stratosphere prototype. A non-tuning operator, e.g., Reduce, Match, DataSink, and CoGroup, can have up to two sorters. Both the sorter and non-adaptive operator can access the memory manager and the IO manager. In our approach (Figure 4.2) of memory tuning prototype of Stratosphere, both the non-adaptive operator and memory adaptive operator are subclasses of the same class monitored operator. The monitored operator has a monitoring system to observe the running state of all operators. All memory adaptive operators are subclasses of a tunable operator, which is generalized from the monitored operator. A monitored operator stores all runtime data observed and measures the size of input datasets, memory allocation and configurations for the operator, its components and the timestamps for check points. The tunable operator implements the interfaces for memory tuning, e.g., the interfaces to estimate the running time, the ROC value, etc. The memory adaptive operator is generalized form the tunable operator, which performs the memory tuning in the operator. The monitored operator provides the method to access the IO manager through the class local IO manager and the memory manager via the class local memory manager (LMM). The type of memory manager is encapsulated behind the LLM interface. The LLM picks the correct interface to allocate and release memory pages according which kind of memory manager is running by Stratosphere. Therefore both the non-adaptive operator and memory adaptive operator work with both kinds of memory manager. The non-tuning operators have only one operation `allocate()` to allocate more memory, which throws an exception if the memory manager has no

![Figure 4.1: The solution of Stratosphere](image)
4.1. IMPLEMENTATION

Figure 4.2: The solution of memory tuning for Stratosphere

more memory to allocate. In this case, the operator and its sorters are aborted. Both the operator and sorter allocate memory independently, without connection between them.

A memory adaptive operator allocates the initial memory by the allocate() defined in the monitored operator. The tuning memory manager detects that it is a new operator. If there is insufficient memory for the operator, it uses a memory tuning mechanism to ask other operators to release memory for the new operator. After the operator allocates the initial memory, the operator can use the getMoreMemory() operation to allocate memory pages. The LLM will invoke the memory tuning interfaces for tuning memory manager; otherwise it will use the allocate(). For memory adaptive operators, it returns empty set of memory pages rather than throwing an exception; therefore, the memory adaptive operators can continue to execute. Sorter in Stratosphere is an independent component, but in our solution it is a class monitored component associated with the monitored operator such as Reduce, Match, etc. A monitored operator can own one or two monitored components, and a tunable operator can own one or two tunable components.

The memory tuning sorter is generalized from tunable component that has connections to the tunable operator. It cannot allocate and release memory itself, but rather through the interface getMoreMemoryForComponent() or returnMemoryToOperator(). Whether the memory pages is allocated or released depends on the decision of the tunable operator, which can adjust the memory allocation between tunable components and itself. By both the non-adaptive and memory adaptive operator, memory allocating and releasing of a sorter is via the LLM, whereby the total allocation of both kinds of operators is recorded by the LLM.

The basic algorithm of both non-adaptive operators and memory adaptive operators are similar. The algorithm of memory adaptive operators manages its memory allocation according to the workloads. The unnecessary memory pages are released, when the algorithm detected them. The minimal memory demand is always protected. Although the minimal memory is defined by the operator implementation, the actual minimal memory can be larger than the defined value. For
example, given the minimal memory, a Hash Join may be aborted for its high recursive partitioning level. The algorithm of memory adaptive operator will start a rescue memory tuning to avoid this situation, whereby the ROC value will be set as the highest value to compete with other operators. The heuristic behavior will postpone the problematic executions until the memory shortage is resolved or it has to be aborted.

We implemented several operators into memory adaptive operators based on tunable operator: adaptive-NLJ, adaptive-BNLJ, adaptive-HJ, adaptive-SMJ, adaptive-Reduce, adaptive-CoGroup, adaptive-DataSink. All operators also used the default memory manager. In this case, the memory tuning functions is turned off by the LMM.

4.2 Experiments

4.2.1 Experimental Setup

We ran the experiments on a compute node with two Intel Xeon E5-2620 2 GHz Processors, each of which has 6 cores. 24 GB main memory is available. The Java Runtime Environment we execute the experiments on is OpenJDK 64-Bit Server VM (build 24.51-b03) on the Operation System Ubuntu 14.04.1 LTS. The memory tuning component is built upon the Stratosphere version 0.2.

We execute each test five times and measure the execution time with the system timestamps for each run. The Java Virtual Machine (JVM) is a code interpreter. We observed that the first run of each experiment, which uses interpreted code, is 5% slower than the compiled code on average. The just-in-time compiler (JIT-compiler) compiles the program frequently executed. Therefore, we drop the first 1 or 2 experiments result for each test and accept only the average value of stable results by compiled codes.

The test datasets are generated by the TPC-H 2.17.0 benchmark. We generate three workloads for different tests. We set the scale factor (SF) as 1, 5, 50, to generate 1G, 10G and 50G populated workloads. Each workload contains eight tables. For the performance test of tuning and non-tuning operators, we use a 1G workload to ensure that we can observe the performance when the workload fits into the memory. For the TPC-H Query 1, 14 and 19 we use the 50G workload to benchmark the system. The complex TPC-H Query 7 produces a large intermediate values by 50G workload, whose size exceeds the capacity of the hard disk, therefore we use a 5G workload to ensure the experiment has successfully finished successfully. For the multi-thread tests, we use 1G workloads to build an operator with a small and medium size workload. For the operator with a large workload we use the 5G workload.

4.2.2 Performance Test of Memory Adaptive Operators

In this subsection, we compare the adaptive operators and the non-adaptive operators. In this test, we run instance of operators one by one. The memory tuning is enabled at the memory manager interface. The total available memory was changed as an evaluation parameter in the memory manager. Initially, the available memory was entirely allocated by the operator, hence leaving no memory for tuning. In this setting, we evaluated the behavior of operators with different memory allocation.

The goal of the performance test is to show the potential of memory tuning. When the memory allocation is changed, the performance is jumped from one point to another in the plot for the rest execution with a memory management costs. The potential improving space of memory tuning is the execution time gap between the minimal memory and the in cache memory size.
4.2. EXPERIMENTS

Figure 4.3: Performance Test of Nested Loop Join

Nested Loop Join and Block Nested Loop Join

We evaluated the Nested Loop Join and Block Nested Loop Join using the `part` and `supplier` table of the TPC-H benchmark at a scale factor of $SF = 1$.

We implemented a FPF replacement to apply a MRU like replacement algorithm to improve the performance of Stratosphere’s repeating reading. However, the expected performance did not show in the Figure 4.3, indeed; in some cases, the performance is even worse. Accordingly, we inspected this unexpected situation. The FPF worked well when the inner dataset is repeated by first a few times. After that, the Operation System cached the file in memory, so the IO cost of repeated reading is almost zero. In this case, the FPF increased the costs of memory management. When less memory was assigned to the NLJ, most pages were loaded from the file cache of the OS. The more memory pages that were assigned to the NLJ, the more records were read from the memory, and the less from the file cache. The mixed reading strategy did not reduce the overall IO cost until the dataset fitted into the memory. We conducted a comparative test on a Windows 7 system, whereby the IO costs of memory adaptive NLJ reduced by more memory allocation, and the IO cost of non-adaptive operator had a rare improvement until the inner dataset was cached in memory. The effect brought by the OS should be investigated in further work.

The implementation of memory adaptive BNLJ was similar to the non-adaptive BNLJ, whereby we inserted pages to the buffer pool of the BNLJ, and thus the management of BNLJ is very small. The result of performance test of BNLJ was in Figure 4.4. The performance of the BNLJ was seldom changed by the memory allocation. As with the NLJ, the file cache of the OS provided a higher performance than a special buffer pool for the BNLJ. The memory tuning effect only little to the NLJ and BNLJ by Linux OS, more memory did not improve the performance of NLJ and the BNLJ; indeed, there was even a negative effect, unless the workload was cached. In the special case of Stratosphere, the workload was known after the dataset of inner loop was loaded into the operator; however, the workload of the dataset of the outer loop was known by the operator when the operator was finished. The system should set more memory to a NLJ or BNLJ only when
it is clear that the inner dataset can be cached; otherwise, they should have a minimal memory allocation.

Hash Join

We evaluated the Hash Join using the lineitem and order table of the TPC-H benchmark at a scale factor of $SF = 1$.

The memory adaptive Hash Join reported a similar performance to the non-adaptive Hash Join. Figure 4.5 showed that the execution time in relation to available memory in the execution time rapidly decreased when the memory allocation avoided the recursive partitioning. When the Hash Join ran in one pass, the algorithm attempted to cache as many partitions as possible and the execution time sank slightly when more partitions were cached.

The unnecessary memory was released immediately before the probing phase, and it always tried to allocate more memory from the memory manager. The execution time of the adaptive Hash Join was longer than the non-adaptive Hash Join with every initial memory allocation approximately less than 0.5 seconds.

Reduce and DataSink

Both Reduce and DataSink have only one input and implement a sorter inside. Reduce aggregates the input dataset, whereas the DataSink scans the sorted dataset. In this test the input dataset was the order table from the TPC-H benchmark with the $SF = 1$. The records were sorted and scanned once; given that there are no differences between Reduce and DataSink in the test, we skipped the latter.

Figure 4.6 showed the execution time in relation to the available memory. The memory adaptive Reduce has more memory management costs than the non-adaptive Reduce. However, the memory adaptive Reduce caches sorted data in cache to reduce the time cost before the memory allocation
4.2. EXPERIMENTS

Figure 4.5: Performance Test of Hash Join

Figure 4.6: Performance Test of Reduce
reaches 500 pages. After 500 pages the Reduce operator changed its strategy of building the memory blocks for sorting, so there were jumps in the curves. After the workload fits into the memory after 3000 pages, the execution time of memory adaptive Reduce increased because it will release all unnecessary memory. Indeed, the more unnecessary memory, the longer the execution time, while the non-turning operator kept the unnecessary memory until the end.

**Sort Merge Join**

In this experiment we used the lineitem table and the order table from the TPC-H benchmark with $SF = 1$. Both tables were sorted and the records with the same order key were grouped together.

At begin of this experiment (Figure 4.7), the memory allocation was very small compare to the workload. The records cached in the memory also comprise a very small portion of data being cached. The time cost of memory management cost more than the time saved by caching data. Until 1,000 pages the memory tuning SMJ had a better performance than the non-adaptive SMJ. When 6,000 pages were assigned to the join, the sorter for order table was in cache, and the memory exceeding in cache size was the amount is shifted to another sorter to sort lineitem. In Figure 4.7, the performance of the memory tuning SMJ was improved by this shifting.

**CoGroup**

For the experiment of CoGroup, we used the same workload as the Sort Merge Join. The different was that the CoGroup operator groups the records with the same key, so it scans the outputs of the both sorter only once, whereas the Sort Merge Join conducts a nested loop to join the records with same key. This means that the UDF second order function used here is much simpler than the Sort Merge Join.

The CoGroup in the experiment (Figure 4.8) had two sorters, whereby the memory pages
4.2. EXPERIMENTS

Figure 4.8: Performance Test of CoGroup

were equally assigned to both sorters. However, the size of the two input datasets were different. Using a memory size of 3,000 pages, order table fits completely into memory fits into memory. The memory adaptive CoGroup will use the released memory from the sorter for order table for the larger lineitem table. The execution time is thus much shorter after about 6,000 pages. After the memory allocation reached 25,000 pages, both relations fitted into the memory, because the memory adaptive CoGroup reuses the memory from the sorter for order table, it reached in cache earlier than non-adaptive CoGrop, but after that it also has a large time cost to release the unnecessary pages.

4.2.3 Assorted Queries of the TPC-H Benchmark

In this subsection we present the benchmark result of the prototype Stratosphere (non-adaptive version) and our approach (memory adaptive version) with TPC-H. We chose four queries from the TPC-H benchmark, namely Query 1, 7, 14, 19. We ran each query five times, and computed the average over the execution times. The execution time began when the first record was read until all instance of operators was finished.

TPC-H Query 1

TPC-H Query 1 is a simple query (Appendix B.1), which refers to one table. The PACT Program of Query 1 is shown in Figure 4.9. The data record is projected and subsequently aggregated by Reduce operator to compute the sum. In this experiment we want to show the performances of Stratosphere and our approach with a simple query. For this query, we used $SF=1$.

In this experiment, the projection was conducted while data records were reading from the table. The only memory consuming operator was Reduce. We set the degree of parallelism of the program to 4, i.e. the Stratosphere started four instances of Reduce operator to process the input dataset parallel. In this experiment, the average execution time of non-adaptive version was
1614.693 seconds, and the average execution time of our memory adaptive version was 1450.337 seconds, meaning that the memory adaptive version ran 2.73 minutes (10.1%) quicker than the non-adaptive version. In this experiment, the memory-adaptive version cached all records in memory, whereas the non-adaptive version paid about 595,966 pages IO reading and writing, whereby the average total IO cost was 93.949 seconds. The memory adaptive version uses a 2-way merge to build the final output before it emitted records. The non-adaptive version used a m-way merge to build the final output, although it grouped the records with same key when the last group was consumed by the UDF reduce function.

The figure 4.10 shows the memory allocation diagram of one of the instances of Reduce. The non-tuning Reduce ran with the initial memory allocation until the end. The TPC-H Query1 had a biased workload. When the key l_returnflag and l_linestatus was distributed into instance of Reduce, some instance did not receive any and thus were finished very early. By memory tuning team, the memory was reused to the instances who had records received. In Figure 4.10, the memory tuning Reduce allocated more memory by a workload discovery mechanism. When all records were sorted, the unnecessary memory was released.

This experiment demonstrates the flexible memory allocation of our approach. Unused memory was reallocated instead of wasting it for finished instances in the non-adaptive version. The memory tuning mechanism led to an improvement in execution time.
TPC-H Query 7

TPC-H Query 7 is a complex query with five tables (appendix B.2). Because this query generates a very large intermediate records, we chose a smaller workload for this query, TPC-H benchmark $SF = 5$. The PACT Program of Query 7 is shown in Figure 4.11. In this program, a Cross operator was invoked to produce the Cartesian product of the table `nation` with itself. Because this table was very small, we used the degree of parallelism 1. For other operators, we chose the degree of parallelism to 4.

Figure 4.12 showed an instance of Match in the Query 7, which was marked in cyan in Figure 4.11. The memory adaptive Match allocated additional memory by workload discovery. After 145 seconds, the system asked it release memory for the following Reduce operator. It merged seven sorted cached data blocks and spilled onto the disk, whereby 65,844 pages were spilled in 32.085 seconds. For the same Match, the non-adaptive Match spilled 272,064 pages in 43.826 seconds. The execution time of the memory adaptive version was 2333.349 seconds, whereas the non-adaptive version cost 2712.589 seconds; thus, the execution time of the memory adaptive version was 14% shorter.

The average execution time of the memory adaption version was 53.055 minutes and the non-adaptive version was 60.716 minutes, meaning the memory adaptive version took 7.661 minutes (12.7%) less time. The memory tuning mechanism reduces the IO cost significantly, and the most data was successfully reduced to 2.74%. Due to the waiting time for the records delivering for the following operators, the total IO cost of non-adaptive operators was shown in the total time cost.

To demonstrate the execution time wasted by waiting for other instances of the same operator, we ran a smaller workload $SF = 1$ with 512M limited total memory, in Figure 4.13. In this figure, the operator has a very short waiting time. The memory adaptive operator allocates more memory by workload discovery and the spilled 3518 pages for other operators. The overall execution time was 7.081 seconds, while the non-adaptive operator had a fixed memory allocation with an execution time of 21.220 seconds, namely almost three times longer. Another Figure (appendix C.6) shows an operators that was asked to release memory at the last memory tuning point short before the output. This memory tuning led to a worse execution time than the non-adaptive operator. If the cost of outputting record is known, releasing memory after this operator was accomplished.
Figure 4.12: Sort Merge Join in TPC-H Query 7

Figure 4.13: Small scale Sort Merge Join in TPC-H Query 7
4.2. EXPERIMENTS

Figure 4.14: Hash Join in TPC-H Query 14

represented the optimal plan.

**TPC-H Query 14**

TPC-H Query 14 is a simple query with two tables. The PACT Program of Query 14 is shown in Figure 3.1 as well as the query List B.3. It contains one Match operator and a Reduce operator. The table \texttt{lineitem} and \texttt{part} are joined, before Reduce is used to aggregate the \texttt{promo_revenue}. For this Query 14 we used the TPC-H benchmark $SF = 50$. The degree of parallelism was set to 4.

In the experiment, Stratosphere chose a Hash Join for the Match operator. The \texttt{part} table is relatively small, so the dataset fits into memory when the part table is partitioning. The Hash Join ran in cache. In Figure 4.14, it is shown that the memory adaptive HJ released the unnecessary memory.

The Reduce operator reused them with memory tuning mechanism, whereby the outputs of the Match operator was successfully cached in memory. However the loading phase lasted about 1330 seconds. In Figure 4.15 the first time the memory adaptive Reduce decreased its memory allocation was shortly after the loading phase. The merging of the memory adaptive Reduce was an in-memory sort. The cache strategy implemented in memory adaptive Reduce produces allocated memory to build empty blocks for sorting. In Figure 4.15, each time the memory allocation of memory adaptive Reduce jumped, the allocated memory was used to build a new memory block for sorting. Before the merging phase, it also released about 20422 pages. The non-tuning Reduce was in one pass. In this case, the memory adaptive Reduce consumes more memory than the non-tuning Reduce, although there are no other operators asked for memory; otherwise, the memory allocation of memory adaptive Reduce should be reduced.

The average execution time of non-tuning team was 1399.134 seconds, whereas the memory tuning team took 1368.890 seconds, thus making the memory tuning team 30.2 seconds faster. The slow arriving rate of the input of Reduce allowed the system significant time to write spilled
pages. By our IO measurement introduced in section 3.2, the IO cost for spilling data was very small; therefore, the non-adaptive version had an average IO reading and writing of 107,294 pages, although the average over the total IO time cost was 4.074 seconds.

TPC-H Query 19

The Query 19 was a simple query with two tables textttlineitem and part (Appendix B.4). The PACT Program of Query 19 is shown in Figure 4.16. It contains three Match operators and a Reduce operator. The three different Match operators worked on the similar workload. The outputs of three Matches were aggregated by a Reduce operator. The degree of parallelism we set as 4, meaning that there were 12 instances of Match and 4 instances of Reduce were running. For this query we used $SF=50$.

In this experiment, the Stratosphere chose the Hash Join for the Match whereby table part was
the building side and the textttlineitem was the probing side. The choice of Stratosphere cached all instances of operators in cache. The average execution time of non-adaptive version was 1286.694 seconds and the memory adaptive memory version was 1288.013 seconds. There were no significant differences between them. For the in cache instance of operators, the only work of memory adaptive version to improve the performance involved releasing the unnecessary memory pages.

4.2.4 Summary

In this section, we first tested the performance of single memory adaptive operators and the non-adaptive operators, before using the assorted queries from TPC-H benchmark to test our version of Stratosphere and the standard version. The memory adaptive operators managed their memory as expected, apart from the Nested Loop Join and the Block Nested Loop Join, which rarely respond to the changing memory allocation. The memory tuning for Nested Loop Join and Block Nested Loop Join is unnecessary by the OS, which caches repeatedly accessed files automatically. The memory management inside of memory adaptive operators is optimized for the overall performance. When the memory management is already optimal, our aim is to use less memory to reach the similar performance by releasing the unnecessary memory pages, e.g. the memory management for Hash Join. We also find opportunities to use allocated memory to caching data and reconfigure them for another usage to reduce the execution time, e.g. Reduce, Sort Merge Join and CoGroup. The initial memory allocation is important; if the system made a mistake, the memory tuning mechanism corrected it. However, the correction is not for free; for instance, if a small operator is assigned a large memory, the costs to free all unnecessary memory pages can also strongly effect the execution time.

In the following part of the section we ran Stratosphere with TPC-H queries. Query 1 showed the situation of a biased input dataset. The memory tuning detected this situation and the memory exchanging reduced the overall execution time. Query 7 provided the system with a complex workload, whereby the memory tuning version also successfully reduced the IO costs as well as the execution time. Query 14 was a special case that the input of the last operator had a very slow arriving rate, the memory tuning version cached all records in memory, while the standard version invokes a lots of IO operation. However, the short execution time after loading all input dataset did not show the privilege of memory tuning. Query 19 is a query with light workload. The executions of both versions were in memory, and thus there were no significant differences.

4.3 Summary

In this chapter, we have presented the design of our memory tuning version of Stratosphere. In section 4.1, we first presented how the tuning memory manager was generalized from the default memory manager. Subsequently we described how memory consuming operators and components were rebuilt and the interactions between either tuning memory manager or the default memory manager. In section 4.2, we executed both versions of Stratosphere for comparison, comparing the performance of memory adaptive operators and non-adaptive operators to show the potential of memory tuning for each operator. Subsequently we ran 4 TPC-H queries to show the performances of both systems.
Chapter 5

Conclusion

5.1 Summary

In this thesis, we have investigated a memory tuning mechanism with Stratosphere, which is a big data analytic platform that has to deal with unknown user defined join functions. For each elementary instance of operator, the size of input is also unknown. By lacking basic information of the workloads, the memory management is therefore a problem. We enhanced the Return on Consumption model of Philip S. Yu and Douglas W. Cornell to evaluate the memory utilization to optimize the memory allocation of all instances of operators. We introduced asynchronous memory tuning mechanism to enable operators to exchange memory pages to improve the global performances. To cope with the changing memory allocation we provide a memory adaptive version of operators, which can adjust their execution parameters with changing memory allocation.

We implemented a memory tuning version of Stratosphere using the Java programming language and evaluated the new version of the system with assorted queries from the TPC-H benchmark. The experiments showed that the memory tuning reused the unnecessary memory of an operator for other operators. The operators with a high value of memory utilization can gain more memory from the operators with lower value. The IO costs of the memory tuning version is significantly less than the standard version, whereby unnecessary memory is immediately released for other operators when it is detected. The free memory is also used to improve the performance of Stratosphere.

5.2 Open issues

There remain open issues for further investigation. The file caching strategy of OS affects the performance of buffering, whereby implementation of memory tuning Nested Loop Join cannot improve its performances. The replacement algorithm has theoretically less IO operations, which is confirmed in the evaluations. However, the reduction of IO cost does not reflect a better performance, given that even more data is conducive to a worse performance. Further investigation may provide an explanation for this observation.

To avoid the interplay of operators, we used the memory manager as a broker. This design simplified the problem by making the operators transparent to each other, but the contacts between operators can also reduces the resource used for queries. In the evaluation, the memory adaptive operators do not show their privilege by wasting time for waiting. If an operator has a very slow arriving rate input from other operators, a better buffering strategy can be used to buffer the input records first or write them onto disk, rather than starting the operator to receive the records by occupying many resources.
The horizontal contacts between instances of one operator are also important to improve the overall performance. If instances of operators have to emit the first output record after all instances are ready, the execution time of operators is affected by the slowest instances. If those instances contact each other, the faster operator can spill data onto the disk and release the memory earlier; in the experiments, some instances of operators occupied memory for waiting in a long time. This memory consumption can be avoided.

The strategy of memory exchanging can also be improved by the global information. In the test, we found that some memory adaptive operators ran faster than non-adaptive operators, but just before emitting output it is busy releasing memory for tuning. The execution time is thus slower than the non-adaptive operators. If the output emitting costs significant time, the memory tuning may improve the overall performance by gain higher performance by other operators. If the output time is very short, this memory tuning wastes time, because after finishing the workload, all memory occupied by this operator is freed. Through knowledge about the instances of the same operator and the details of the channel used to connect with the following operators, better decisions can be made to avoid incorrect memory tuning decisions.
Appendix A

Algorithms of Memory Adaptive Operators

A.1 Algorithm of Adaptive Nested Loop Join

Algorithm A.1 Algorithm of Adaptive Nested Loop Join

1: function ADAPTIVENESTEDLOOPJOIN((R, S))
2: spilled = false
3: for r ∈ R do
4:     if insert(buffer, r) = false then
5:         if EXTENDBUFFER(buffer) = false then
6:             spill(buffer)
7:             spilled = true
8:         end if
9:         insert(buffer, r)
10:     end if
11: end for
12: if spilled = true then
13:     spill(buffer)
14: end if
15: count = 0
16: for s ∈ S do
17:     count = count + 1
18:     if count = 100 then
19:         MEMORYTUNING(buffer)
20:     end if
21: end if
22: for r ∈ buffer do
23:     if Join(r, s) = true then emit(< r, s >)
24: end if
25: end for
26: buffer.reset()
27: end function
Algorithm A.2 Algorithm of Adaptive Block Nested Loop Join

1: function AdaptiveNestedLoopJoin((R, S))
2:  spilled = false
3:  for r ∈ R do ▷ load inner
4:      if insert(buffer, r) = false then
5:          if extendBuffer(buffer) = true then ▷ workload discovery
6:              insert(buffer, r)
7:          else
8:              spill(buffer)
9:              spilled = true
10:             insert(buffer, t)
11:         end if
12:     end if
13:  end for
14:  if spilled = true then ▷ spill all data, save a copy
15:      spill(buffer)
16:  else
17:      convertToNLJ() ▷ Convert to NLJ to avoid IO cost
18:  end if
19:  memoryPages = buffer.getMemory ▷ Get memory to build a buffer for outer.
20:  memoryBuffer(memoryPages) ▷ Build in memory buffer for outer.
21:  for s ∈ S do
22:      memoryBuffer.insert(s)
23:      if memoryBuffer is full then
24:          MEMORY_TUNING(buffer, memoryBuffer)
25:      end if
26:      if memoryBuffer is full then
27:          for r ∈ buffer do
28:              for s ∈ memoryBuffer do
29:                  if Join(r, s) = true then emit(<r, s>)
30:              end if
31:          end for
32:      end for
33:  end for
34:  if memoryPages = ∅ then
35:      b.release(memRequest)
36:  else if buffer.memory + memoryBuffer.memory ≥ |R| then ▷ Convert to NLJ to avoid IO cost
37:      convertToNLJ()
38:  end if
39:  end if
40: end function

Algorithm A.3 Algorithm of Workload Discovery for NLJ and BNLJ

1: function extendBuffer(b)
2:  if b.spilled = false then
3:      memoryPages = workloadDiscovery()
4:  if memoryPages = ∅ then return false
5:  end if
6:  b.insertMemory(memoryPages)
7:  end if return false
8: end function
Algorithm A.4 Algorithm of Memory Tuning for NLJ

1: function memoryTuning(b)
2:   if b.spilled = false then \(\triangleright\) all records are buffered, no tuning
3:     return
4:   end if
5:   memoryPages = memoryManager.tuning(b.size – b.memory)
6:   if memoryPages = ∅ ∧ memRequest > 0 then \(\triangleright\) loser, release
7:     b.release(memRequest)
8:   end if
9:   b.insertMemory(memoryPages) \(\triangleright\) winner, insert memory
10: end function

Algorithm A.5 Algorithm of Memory Tuning for BNLJ

1: function memoryTuning(b)
2:   if b.spilled = false then \(\triangleright\) all records are buffered, no tuning
3:     return
4:   end if
5:   memoryPages = memoryManager.tuning(b.size – b.memory)
6:   if memoryPages = ∅ ∧ memRequest > 0 then \(\triangleright\) loser, release
7:     return
8:   end if
9:   b.insertMemory(memoryPages) \(\triangleright\) winner, insert memory
10: end function
A.2 Algorithm of Adaptive Hash Join

Algorithm A.6 Algorithm of Adaptive Hash Join

1: function AdaptiveNestedLoopJoin(R, S) ▷ building phase
2: for r ∈ R do
3:   hashtable.insert(r)
4:   if hashtable is full ∧ hash has not spilled then
5:     WorkloadDiscovery(hashtable) ▷ if successful, pages are inserted
6:   else if hashtable is full then
7:     memoryTuning(hashtable) ▷ pages are inserted or are released by spilling
8:     end if
9:   while hashtable is full do
10:     hashtable.spillPartition()
11:   end while
12: end for
13: for s ∈ S do ▷ probing phase
14:   l = hashtable.probe(s)
15:   if l ≠ ∅ then
16:     if Join(l, s) = true then
17:       emit(<l, s>)
18:     end if
19:   else if s hits a spilled partition then
20:     hashtable.put(s)
21:   end if
22: end for
23: delete in memory partition
24: while hashtable has spilled Partition do
25:   if memoryAllocation > size of biggest spilled partition then
26:     release unnecessary memory
27:   else if memoryAllocation < size of biggest Spilled partition then
28:     memoryTuning(size of biggest Spilled partition − memoryAllocation)
29:   end if
30: R = hashtable.nextSpilledPartition()
31: S = R.getProbeRecord()
32: for s ∈ S do
33:   l = hashtable.probe(s)
34:   if l ≠ ∅ then
35:     if Join(l, s) = true then
36:       emit(<l, s>)
37:     end if
38:   end if
39: end for
40: end while
41: end function

A.3 Algorithm of Adaptive Sorter
Algorithm A.7 Algorithm of Adaptive Sorter

1: \( \text{blocks} = \emptyset \)
2: \( \text{function LoadingThread}(R) \)
3: \( \text{block} = \text{getNextFreeBlock}() \)
4: \( \text{for} \; r \in R \text{ do} \quad \triangleright \text{building phase} \\
5: \quad \text{block}.\text{insert}(r) \\
6: \quad \text{if} \; \text{block is full} \text{ then} \\
7: \quad \quad \text{send the block to sorting thread} \\
8: \quad \quad \text{block} = \text{getNextFreeBlock}() \\
9: \quad \quad \text{if} \; \text{no more free block} \text{ then} \\
10: \quad \quad \quad \text{send a spill signal to spilling thread} \\
11: \quad \quad \text{end if} \\
12: \quad \text{end if} \\
13: \text{end for} \\
14: \text{send the block to sorting thread, send EOF to sorting thread} \\
15: \text{end function} \\
16: \text{function SortingThread} \\
17: \text{receive block } b \text{ from loading thread} \\
18: \text{quicksort}(b) \\
19: \text{send } b \text{ to spilling thread} \\
20: \text{end function} \\
21: \text{function Spilling Thread} \\
22: \text{spill} = \text{true}, \text{eof} = \text{false}; \\
23: \text{while} \; \text{receive a block } b \text{ from sorting} \land \text{spill} = \text{false} \land \text{eof} = \text{false} \text{ do} \quad \triangleright \text{In Cache} \\
24: \quad \text{blocks} = \text{blocks} \cup b \\
25: \quad \text{if} \; \text{receive a spill signal} \text{ then} \\
26: \quad \quad \text{moreMemory} = \text{workloadDiscovery}() \\
27: \quad \quad \text{if} \; \text{moreMemory} \neq \emptyset \text{ then} \\
28: \quad \quad \quad b = \text{buildBlock}(\text{moreMemory}) \text{ and send } b \text{ to loading thread} \\
29: \quad \quad \text{else} \\
30: \quad \quad \quad \text{spill} = \text{true} \\
31: \quad \quad \text{end if} \\
32: \quad \text{end if} \\
33: \quad \text{if} \; \text{receive EOF} \text{ then} \\
34: \quad \quad \text{eof} = \text{true} \\
35: \quad \text{end if} \\
36: \text{end while} \\
37: \text{while} \; \text{eof} = \text{false} \text{ do} \quad \triangleright \text{receiving blocks and spill} \\
38: \quad \text{if less than two block in loading and sorting thread} \text{ then} \\
39: \quad \quad \text{moreMemory} = \text{memorytuning}() \\
40: \quad \quad \text{if} \; \text{moreMemory} \neq \emptyset \text{ then} \\
41: \quad \quad \quad b = \text{buildBlock}(\text{moreMemory}) \text{ and send } b \text{ to loading thread} \\
42: \quad \quad \text{else} \\
43: \quad \quad \quad \text{spill}() \\
44: \quad \quad \quad \text{if} \; \text{memoryRequest} > 0 \text{ then} \\
45: \quad \quad \quad \quad \text{deconstruct some blocks and free the memory} \\
46: \quad \quad \quad \text{end if} \\
47: \quad \quad \text{send all blocks in } \text{blocks} \text{ to loading thread} \\
48: \quad \quad \text{blocks} = \emptyset \\
49: \quad \text{end if} \\
50: \quad \text{end if} \\
51: \text{if} \; \text{receive a block } b \text{ then} \\
52: \quad \text{blocks} = \text{blocks} \cup b \\
53: \text{else if} \; \text{receive a eof from loading thread} \text{ then} \\
54: \quad \text{eof} = \text{true} \\
55: \text{end if} \\
56: \text{end if} \\
57: \text{end while} \\
58: \text{return } \text{merge}(\text{mergedInDisk}, \text{mergedInMemoryBlock}); \\
59: \text{end function}
Appendix B

The SQL Query of TPC-H Benchmark

B.1 TPC-H Query 1

```sql
select
    l_returnflag,
    l_linestatus,
    sum(l_quantity) as sum_qty,
    sum(l_extendedprice) as sum_base_price,
    sum(l_extendedprice*(1-l_discount)) as sum_disc_price,
    sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge,
    avg(l_quantity) as avg_qty,
    avg(l_extendedprice) as avg_price,
    avg(l_discount) as avg_disc,
    count(*) as count_order
from
    lineitem
where
    l_shipdate <= date '1998-12-01' - interval '[DELTA]' day (3)
group by
    l_returnflag,
    l_linestatus
order by
    l_returnflag,
    l_linestatus
```

Listing B.1: SQL TPC-H Query 1

B.2 TPC-H Query 7

```sql
select
    supp_nation,
    cust_nation,
    l_year,
```

Listing B.2: SQL TPC-H Query 7
APPENDIX B. THE SQL QUERY OF TPC-H BENCHMARK

\[
\text{sum(volume) as revenue from (}
\begin{align*}
\text{select} & \quad n1.n\text{.n_name as supp_nation,} \\
& \quad n2.n\text{.n_name as cust_nation,} \\
& \quad extract(year from l\text{.shipdate}) as l_year, \\
& \quad l\text{.extendedprice} \times (1 - l\text{.discount}) as volume \\
\text{from} & \quad supplier, \\
& \quad lineitem, \\
& \quad orders, \\
& \quad customer, \\
& \quad nation n1, \\
& \quad nation n2 \\
\text{where} & \quad s\text{.suppkey} = l\text{.suppkey} \text{ and o\text{.orderkey}} = l\text{.orderkey} \\
& \quad \text{and c\text{.custkey}} = o\text{.custkey} \text{ and s\text{.nationkey}} = n1.n\text{.nationkey} \\
& \quad \text{and c\text{.nationkey}} = n2.n\text{.nationkey} \\
& \quad \text{and (}
\begin{align*}
& \quad (n1.n\text{.name} = '[\text{\textit{NATION1}}]' \text{ and n2.n\text{.name}} = '[\text{\textit{NATION2}}]' ) \\
& \quad \text{or (n1.n\text{.name} = '[\text{\textit{NATION2}}]' \text{ and n2.n\text{.name}} = '[\text{\textit{NATION1}}]' )}
\end{align*}
\text{and l\text{.shipdate} between date '1995-01-01' and date '1996-12-31'}
\) \text{as shipping}
\text{group by} \\
\text{supp_nation,} \\
\text{cust_nation, l_year}
\text{order by} \\
\text{supp_nation,} \\
\text{cust_nation, l_year;}
\end{align*}
\]

Listing B.2: SQL TPC-H Query 7

B.3 TPC-H Query 14

\[
\text{select} \\
100.00 \times \text{sum(}
\begin{align*}
\text{case} & \quad \text{when p\text{.type} like 'PROMO' then} \\
& \quad l\text{.extendedprice}\times(1 - l\text{.discount}) \\
& \quad \text{else} \\
& \quad 0 \\
\end{align*}
\text{end}) \\
/ \text{sum(l\text{.extendedprice} \times (1 - l\text{.discount}) as promo_revenue from} \\
lineitem, part \\
\text{where} \\
l\text{.partkey} = p\text{.partkey} \\
\text{and l\text{.shipdate} >= date 'DATE'} \\
\text{and l\text{.shipdate} < date 'DATE' + interval '1' month;}
\]

Listing B.3: Example SQL TPC-H Query 14
B.4 TPC-H Query 19

```sql
select sum(l_extendedprice * (1 - l_discount)) as revenue
from lineitem,
part
where (p_partkey = l_partkey and p_brand = '"BRAND1"'
and p_container in ('SM\_CASE', 'SM\_BOX', 'SM\_PACK', 'SM\_PKG')
and l_quantity >= "QUANTITY1" and l_quantity <= "QUANTITY1" + 10
and p_size between 1 and 5 and l_shipmode in ('AIR', 'AIR\_REG')
and l_shipinstruct = 'DELIVER\_IN\_PERSON')
or (p_partkey = l_partkey and p_brand = '"BRAND2"'
and p_container in ('MED\_BAG', 'MED\_BOX', 'MED\_PKG', 'MED\_PACK')
and l_quantity >= "QUANTITY2" and l_quantity <= "QUANTITY2" + 10
and p_size between 1 and 10 and l_shipmode in ('AIR', 'AIR\_REG')
and l_shipinstruct = 'DELIVER\_IN\_PERSON')
or (p_partkey = l_partkey and p_brand = '"BRAND3"'
and p_container in ('LG\_CASE', 'LG\_BOX', 'LG\_PACK', 'LG\_PKG')
and l_quantity >= "QUANTITY3" and l_quantity <= "QUANTITY3" + 10
and p_size between 1 and 15
and l_shipmode in ('AIR', 'AIR\_REG')
and l_shipinstruct = 'DELIVER\_IN\_PERSON');
```

Listing B.4: SQL TPC-H Query 19
Appendix C

Results of TPC-H Query 7
Figure C.1: Example of Block Nested Loop Join in Query 7

Figure C.2: Example of Hash Join CNN in Query 7
APPENDIX C. RESULTS OF TPC-H QUERY 7

Figure C.3: Example of Sort Merge Join SCNN in Query 7

Figure C.4: Example of Hash Join OL in Query 7
Figure C.5: Example of Reduce in Query 7

Figure C.6: Example of small scale Hash Join OL in Query 7
Bibliography


Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe. Weiterhin erkläre ich, eine ...arbeit in diesem Studiengebiet erstmalig einzureichen.

Berlin, den 16th September 2014

Statement of authorship

I declare that I completed this thesis on my own and that information which has been directly or indirectly taken from other sources has been noted as such. Neither this nor a similar work has been presented to an examination committee.

Berlin, 16th September 2014