Elastic Online Analytical Processing on RAMCloud

Christian Tinnefeld[#], Donald Kossmann[*], Martin Grund[#], Joos-Hendrik Boese[+], Frank Renkes[+], Vishal Sikka[+], Hasso Plattner[#]

[#]*Hasso Plattner Institute, University of Potsdam, Germany*
{firstname.lastname}@hpi.uni-potsdam.de
[*]*Systems Group, ETH Zurich, Switzerland*
donald.kossmann@inf.ethz.ch
[+]*SAP AG, Walldorf, Germany*
{firstname.lastname}@sap.com

ABSTRACT

A shared-nothing architecture is state-of-the-art for deploying a distributed analytical in-memory database management system: it preserves the in-memory performance advantage by processing data locally on each node but is difficult to scale out. Modern switched fabric communication links such as InfiniBand narrow the performance gap between local and remote DRAM data access to a single order of magnitude. Based on these premises, we introduce a distributed in-memory database architecture that separates the query execution engine and data access: this enables a) the usage of a large-scale DRAM-based storage system such as Stanford's RAMCloud and b) the push-down of bandwidth-intensive database operators into the storage system. We address the resulting challenges such as finding the optimal operator execution strategy and partitioning scheme. We demonstrate that such an architecture delivers both: the elasticity of a shared-storage approach and the performance characteristics of operating on local DRAM.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Distributed databases, Query Processing*

General Terms

Theory, Performance, Experimentation

Keywords

Analytics, In-Memory, Elasticity, RAMCloud

1. INTRODUCTION

The storage and query processing capacity of an analytical in-memory database management system (DBMS) on a single server is limited. Overcoming this limitation is especially important for analytical applications since they consume large data sets. A shared-nothing architecture enables the combined usage of the storage and query processing capacities of several servers in a cluster. Each server in this cluster is assigned a partition of the overall data set and processes its locally stored data during query execution. This architecture is currently state-of-the-art in the context of distributed analytical in-memory DBMSs [7] as it preserves the performance advantage of main memory data storage by processing data locally on each server: this advantage vanishes in a shared-storage approach where the data has to be constantly shipped over a network whose performance characteristics are multiple orders of magnitude worse in terms of latency and bandwidth than local memory access.

On the other hand, deploying a shared-nothing architecture comes at a price: scaling out such a cluster is a hard task [5]. Chosen partitioning criteria have to be reevaluated and potentially changed. While automated solutions exist to assist such tasks, often manual tuning is required to achieve the desired performance characteristics. Furthermore, such an architecture prohibits the independent scale-out of storage and query processing capacities. The situation becomes even more complex when recovery and availability guarantees need to be considered. The resulting limitations of such an architecture in terms of flexibility and elasticity make the adaptation to frequently changing storage and query processing requirements – e.g. in a cloud environment – difficult.

Recent developments in the area of high-performance computer networking technologies narrow the performance gap between local and remote DRAM data access to and even below a single order of magnitude. For example, InfiniBand [8] specifies a maximum bandwidth of up to 300 Gbit/s and an end-to-end latency of 1-2 µs, an Intel Xeon Processor E5-4650 [9] has a maximum memory bandwidth of 409.6 Gbit/s and a main memory access latency of 0.1 µs. The research project RAMCloud [17] demonstrates that the performance characteristics of a high-performance computer network can be exploited in a large distributed DRAM-based storage system to preserve the narrowed performance gap between local and remote data access at scale.

Based on these premises, we designed a distributed database architecture that separates the query execution engine and the data access. We implemented AnalyticsDB, a prototypical analytical in-memory DBMS, which demonstrates that such a separation allows to plug-in a storage system such as RAMCloud instead of local DRAM. Our experiments in this paper show that a) such an architecture enables AnalyticsDB to leverage RAMCloud's elasticity advantages as RAMCloud can be arbitrarily resized while AnalyticsDB executes queries. In addition, AnalyticsDB processes its queries only 2.6 times slower on RAMCloud than on local DRAM.

EDBT/ICDT '13, March 18 - 22 2013, Genoa, Italy

The architecture of AnalyticsDB also allows to push down the execution of bandwidth-intensive operators into RAMCloud. Our experiments demonstrate that b) this reduces the AnalyticsDB query execution time penalty on RAMCloud to 11% in comparison to an execution on local DRAM. While this measurement is based on a push down of the AnalyticsDB operators to a single RAMCloud node, we illustrate that c) the operator push down allows a parallel execution of AnalyticsDB operators across multiple RAMCloud nodes resulting in an additional acceleration of the query execution.

Consequently, we show that the introduced architecture combines the advantages of a shared-storage approach (maximum elasticity) with the advantages of a shared-nothing approach (maximum performance by executing operations as close to the data as possible) for an analytical in-memory DBMS: the narrowed performance gap between local and remote DRAM access is the enabler of applying such an architecture. Specifically we make the following contributions:

i We introduce the analytical in-memory DBMS AnalyticsDB that defines an API between query execution engine and data access. The API provides two strategies for operator execution: data pull and operator push.

ii Besides the advantages of an operator push execution strategy, there are scenarios where a data pull execution strategy is beneficial. We provide an operator execution cost model to decide on the optimal strategy.

iii We investigate the different data partitioning options in RAMCloud and show the optimal partitioning schemes.

iv We provide an extensive performance evaluation of AnalyticsDB on RAMCloud with the Star Schema Benchmark as workload. We show that AnalyticsDB on RAMCloud preserves the in-memory performance advantage and provides an elastic workload adaption at the same time.

The remainder of the paper is structured as follows: Section 2 presents an overview on the architecture in this paper and describes AnalyticsDB and RAMCloud in more detail. Section 3 explains the involved data storage and partitioning mechanisms and covers aspects such as the mapping from AnalyticsDB columns to objects in RAMCloud. Section 4 describes the identification and implementation of AnalyticsDB operators which are eligible for being pushed into RAMCloud. Section 5 introduces an AnalyticsDB execution cost model that describes the impact of system- and query-related parameters on the operator execution time. It also covers the discussion of data pull vs. operator push execution strategies. Section 6 presents a performance evaluation. Section 7 lists the related work and Section 8 closes with a conclusion.

2. ARCHITECTURAL OVERVIEW

In our system a set of AnalyticsDB nodes access a shared storage layer established by a RAMCloud storage system.

Figure 1 depicts the architectural overview of our system: AnalyticsDB nodes receive application queries dispatched by a central federator node. Every query is assigned to a single AnalyticsDB node, while a local query processor controls its execution. Each AnalyticsDB node holds the meta data describing the relational structure of all data contained in the storage layer to allow for query validation and planning. The query processor accesses the RAMCloud storage system through a RAMCloud client component that transparently maps the AnalyticDB API to operations on specific RAMCloud nodes.

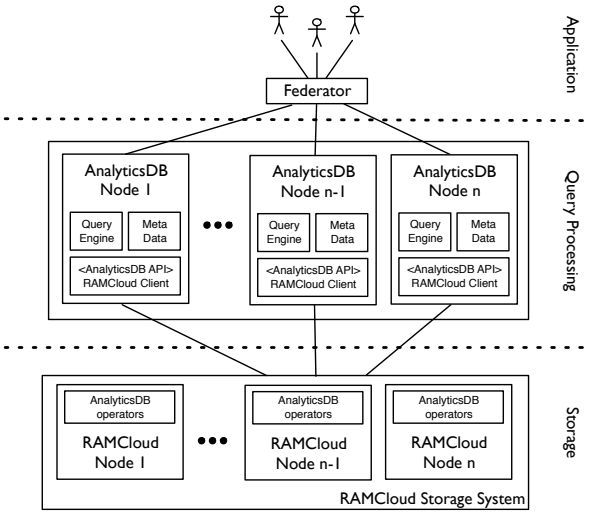


Figure 1: Architectural Overview

The RAMCloud in-memory storage system consists of multiple nodes and takes care of transparently distributing the stored data among participating nodes and manages replication, availability and scaling of the cluster. To allow for push-down of operations to the storage layer, each RAMCloud node is extended with a set of AnalyticsDB operators.

In the following subsections the AnalyticsDB and RAMCloud systems are described in more detail.

2.1 AnalyticsDB

AnalyticsDB is our prototypical analytical in-memory DBMS written in C++. It is built based on the common design principles for online analytical query engines such as MonetDB [2] or SAP HANA [7]. Data is organized in a column-oriented storage format and dictionary compressed. AnalyticsDB uses dictionary compression for non-numeric attributes (e.g. string) and encodes them to int64 values. This results in only integer values being stored in RAMCloud. The dictionary is kept on the AnalyticsDB nodes. The operators of our execution engine are optimized for main memory access and allow to achieve a high scan speed. In addition, our system defers the materialization of intermediate results until it becomes necessary following the pattern of late materialization [1]. The biggest difference from our prototype system to a text-book analytical database system is how the execution engine operates on main memory. Instead of directly accessing the memory, e.g. by using the memory address, we introduced the AnalyticsDB API to encapsulate storage access. This API can be implemented by e.g. using a local data structure or by using the client of a separate storage system such as the RAMCloud client.

AnalyticsDB is designed for online analytical processing (OLAP) allowing data analysts to explore data by submitting ad-hoc queries. This style of interactive data analysis requires AnalyticsDB to execute arbitrary queries on multi-dimensional data in sub-seconds [25]. For performance evaluation we therefore use the Star Schema Benchmark (SSB) [15] [14] as workload later in Section 6. The Star Schema Benchmark data model represents sales data and consists of Customer, Supplier, Part, and Date dimension tables and a Lineorder fact table.

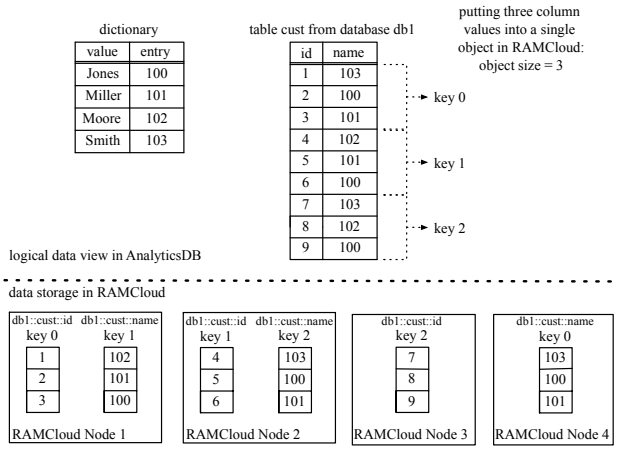


Figure 2: Mapping from AnalyticsDB columns to objects in RAMCloud. Partitioning of two columns across four storage nodes with *server span*=3.

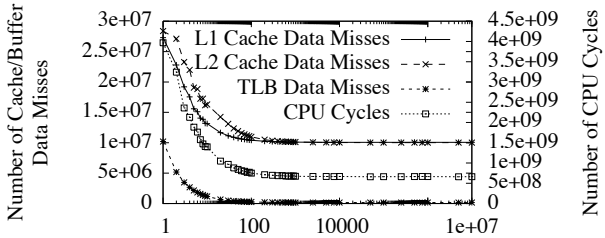


Figure 3: Micro benchmark showing the impact of consecutive memory size and scan performance in main memory for 10 million 8-byte data items. The benchmark shows that the same scan performance can be reached on randomly in memory placed blocks with 1000 consecutive data items each as on a single block with 10 million consecutive data items.

2.2 RAMCloud

RAMCloud is a general-purpose storage system that keeps all data entirely in DRAM by aggregating the main memory of multiple of commodity servers at scale [17]. All of those servers are connected via a high-end network such as InfiniBand which provides low latency [21] and a high bandwidth. RAMCloud employs randomized techniques to manage the system in a scalable and decentralized fashion and is based on a key-value data model: in the remainder of the paper we refer to the data items in RAMCloud as key-value pairs or objects. RAMCloud scatters backup data across hundreds or thousands of disks, and it harnesses hundreds of servers in parallel to reconstruct lost data. The system uses a log-structured approach for all its data, in DRAM as well as on disk; this provides high performance both during normal operation and during recovery [16].

3. STORAGE AND DATA PARTITIONING

While AnalyticsDB logically works on columnar-structured data, our storage layer RAMCloud persists data in a key-value data model. How key-value pairs are allotted to specific RAMCloud nodes determines how column data is partitioned across multiple storage nodes. Since the size of data partitions significantly affects the performance of operators pushed down to the storage system, this process is critical for the overall system performance.

Using the example depicted in Figure 2, we first describe how columns are logically split into key-value pairs and, afterwards, how they are physically stored on RAMCloud nodes. In the end we present a micro benchmark to demonstrate the influence of partition sizes on scan performance.

3.1 Mapping of Column Data to Key-Value Pairs

RAMCloud provides the concept of namespaces. A namespace defines a logical container for a set of objects, where each object key occurs only once. To map an AnalyticsDB table, we create a namespace for each database table attribute with the naming convention *"dbname::dbtablename::attributename"*. In each namespace we create a number of objects, while each object stores a chunk of the corresponding attribute column. How many column values are held by a single object is configurable via a parameter *object size*. The object size parameter and the actual size of the column determine how many objects have to be created for storing the complete column. Figure 2 depicts this concept for a table consisting of two columns *id* and *name* with *object size*=3. To store the complete example table we create a namespace for each attribute and create three objects with keys 0-2 for every column.

3.2 Distribution of Key-Value Pairs in RAMCloud

Upon the creation of a new namespace, the parameter *server span* is set to define how many storage nodes will be used to store the objects of the namespace. These namespaces are assigned to nodes in a round robin manner. Assignment of key-value pairs across nodes is done by partitioning the range of the hashes of the object keys contained in that namespace.

In the example depicted in Figure 2 we define *server span=3* for namespace *"db1::cust:id"* and *"db1:cust:name"* resulting in the shown distribution for a four node RAMCloud cluster.

Putting this partitioning mechanism in context with the aforementioned data mapping has the following implications: the partition granularity is on AnalyticsDB column level. This means it is not possible to enforce placing e.g. an entire AnalyticsDB table that consists of several columns on a single RAMCloud storage node (except when the RAMCloud cluster has only one node).

3.3 Performance Impact of Object Size

A column-oriented data storage provides a fast sequential memory scan speed because of high data locality and thus the possibility to exploit hardware data prefetching. This raises the question how the splitting of columns into small partitions impacts the scan speed. Figure 3 presents a micro benchmark where 10 million 8-byte data items are stored in DRAM and traversed on an Intel Xeon E5450. The x-axis indicates how many of those data items are placed consecutively in DRAM. On the left end of the x-axis is one data item at a time placed consecutively in DRAM. On the right end of the x-axis 10 million data items are placed consecutively which means the complete dataset is stored sequentially in DRAM. The y-axes depict the number of occurring data cache misses and the number of required CPU cycles for completing the traversal. The micro benchmark illustrates that the required number of CPU cy-

Listing 1: Simplified AnalyticsDB Column API

```
ColumnPosition append(ColumnValue value);
ColumnValue get(ColumnPosition position);
void set(ColumnPosition position, ColumnValue value);

ColumnPositionList scan(SCAN_COMPARATOR comparator,
                        ColumnValue value,
                        ColumnPositionList positionList);
Array<ColumnValue> materialize(ColumnPositionList
                               positionList);
ColumnPositionList joinProbe(ArrayRef dimensionTablePKs,
                             PositionListRef
                             validFactTablePos);

size_t size();
void restore(ArrayRef values);
```

cles becomes minimal if a relatively small amount of data items are placed consecutively in DRAM [27] and therefore the maximum scan speed has already been reached.

In the evaluation of our system we choose the allowed upper limit of 1MB for RAMCloud objects which results in an object size of 131.072 (as an AnalyticsDB column value is 8 bytes). Given the results of our benchmark above we conclude that we still achieve maximum scan performance with this partitioning schema: this will be validated in Section 5 in a scan operation micro benchmark shown in Figure 4(a).

4. QUERY EXECUTION AND OPERATOR PUSH DOWN

So far we described how RAMCloud is used as the shared storage layer in our system. With a standard configuration of RAMCloud, query execution can only happen on an AnalyticsDB node by loading the required data from the storage layer into the query processing engine of an AnalyticsDB node. In this section we describe how we extended the RAMCloud system to allow for execution of database operators directly in the storage layer close to the data. Specifically, we first identify which operators are most significant for a database system designed for analytical workloads such as AnalyticsDB and secondly describe how we designed and implemented these operators in RAMCloud.

4.1 Identifying Operators for Push Down

We analyzed the queries of the SSB benchmark to identify the operators which are suitable for and benefit from a push down to the storage layer. Table 1 shows the AnalyticsDB operator breakdown for one execution cycle of the SSB with a scale factor of 10 in local main memory on an Intel Xeon E5620. We normalized the complete execution time to highlight the contribution of each operator to the total execution time and ranked the operators accordingly.

To choose promising operators two aspects are relevant: a) to what extent does an operator contribute to the overall execution time and b) does the operator usually work on data as stored in the persistence layer or on intermediate results. Since we do not want to rebuild a complete query processor in the storage system, but to push-down the execution of stateless and data intensive operators, we do not consider operators for push down that normally work on intermediate result sets.

Table 1 shows that the Hash-Join and Scan operator accumulate 82% of the total execution time in the SSB. From our query execution plans for the SSB we derived that these operators are always the first that touch the raw data and consume it sequentially. The Materialization operator works also directly on the data as stored in the persistence, e.g. when retrieving the actual values in a column based on a position list. Consequently, we decided to implement support for these operators on the storage layer.

Table 1: AnalyticsDB Operator break-down when executing the Star-Schema Benchmark, normalized by the contribution of the operator to the overall query runtime

	Hash-Join	Scan	Group-By	Materialization	Merge-Positions	Sort	Arithmetic
%	0.6657	0.1594	0.0754	0.0693	0.0283	0.0017	0.0003

The Hash-Join operator in AnalyticsDB is optimized for star-schemas and operates on database table level and therefore cannot be entirely executed on the storage level. To support Hash-Join operations on the storage layer we added support for the join probing and materialization sub-operator of the Hash-Join.

A typical join operation in AnalyticsDB joins a fact table with a number of dimension tables and is executed by the AnalyticsDB query engine as follows: first, the join paths are being evaluated using the join probing sub-operator to probe the foreign keys in the respective columns of the fact table (line orders in SSB) against the keys provided in the join path. The keys provided by the join path (primary keys of a dimension table) are normally derived by a preceding scan operation on that dimension table. The evaluation of each join path results in a list of line order positions indicating at which position the join condition is met. This position list acts as additional input for the evaluation of the next join path. The result of the evaluation of all join paths is a list with the fact table positions that fulfill all join conditions. Based on that list, the materialization of the requested attributes for the result table is done. Thus, our Hash-Join implementation accepts three parameters: a) the name of the table that will be joined with n other tables; b) a number of join paths where each join path defines a join with another table by providing a list of foreign keys as join condition; and c) a list of table and attribute names which define the attributes of the result table.

Due to the operation on database table level, the AnalyticsDB Hash-Join operator cannot be pushed down into RAMCloud in its entirety, but instead we push down its join probing sub-operator and materialization sub-operator. During a SSB execution, the execution time ratio between join-probing and materialization is 9.6:1 in the context of a hash-join execution. Consequently, we added support for the *Scan*, *Materialization*, and *Join-Probing* operator in RAMCloud.

The Group-By, Merge-Positions, Sort, and Arithmetic operators work mostly on intermediate results which are processed inside the query engine of an AnalyticsDB node and therefore cannot be pushed down to the storage layer.

4.2 Implementation of AnalyticsDB Operators in RAMCloud

To allow for the push-down of Scan, Materialization, and Join-Probing operators to RAMCloud nodes, we implemented support for these operators in RAMCloud and added the operator signatures to the AnalyticsDB storage API as shown in Listing 1. The *scan*, *materialize*, and *joinprobe* operations operate on column-level and

take a condition and optionally a position list as input parameters. The condition parameterizes the operation itself (e.g. to scan for which values or to hash probe against which probing data). The optional position list defines at which column positions the operator should be executed. If the position list is empty, the entire column will be processed. Depending on the operator, the result of an operator execution is either a list of column values or column positions.

To implement the AnalyticsDB storage API for RAMCloud, we added RAMCloud client code in AnalyticsDB for invoking the operators in RAMCloud. The RAMCloud client component is responsible for mapping the columnar data to RAMCloud namespaces and objects. It resolves what column chunks are stored on which node in the RAMCloud storage system and sends the respective operator input parameters to the relevant nodes. At the RAMCloud node the operator is invoked locally and processes the data specified by the input parameter. In RAMCloud, this specification is either on object granularity (e.g. materialize all values that are contained in the objects *10* and *11*) or on value granularity (e.g. materialize the first three values inside object *10*). Finally, the RAMCloud client component in AnalyticsDB receives the results from all RAMCloud nodes, merges them, and returns the operator result to the query engine on the AnalyticsDB node. The operators have the following properties:

4.2.1 Scan

The Scan operator expects a scan comparator (e.g. less than, equal) and the comparative value. It is possible to provide two comparators and comparative values for doing a range-based scan (e.g. greater than 10 and smaller than 20). Additionally, an optional position list can be passed.

4.2.2 Materialization

The Materialize operator takes a position list as input and returns all values at the defined positions. If no position list is supplied, the Materialize operator returns the values of the the entire column.

4.2.3 Join-Probing

The join probing operator takes the probing data set and a position list to indicate at which positions it should perform the probing. If the position list is empty, it performs the probing on all values in the column. The join probing operator returns a list of positions that indicate where the probing succeeded.

5. ANALYTICSDB EXECUTION COST MODEL

In this section we introduce an execution cost model for AnalyticsDB to analyze the impact of different parameters that have been induced by the data mapping, column partitioning and the design of the operators itself. We first derive an abstract system model which is later used to predict execution costs analytically for different scenarios. Afterwards we use our cost model to evaluate operator push down and data pull execution strategies and show how the cost model can be used to decide on different execution strategies.

5.1 System Model

We abstract the following system model: a column C is defined by the number of contained records S_C and the size of a single record S_r in bytes. It may be partitioned among n RAMCloud nodes $RN_1, \ldots, RN_n$, resulting in disjoint non-overlapping partitions $C_1, \ldots, C_n$ with sizes $S_{C,1}, \ldots, S_{C,n}$. All nodes are connected by network channels with constant bandwidth BW_{Net}, measured in bytes per second.

As described in Subsection 4.2, execution of an operation O for a column C is coordinated by a single AnalyticsDB node AN, while O is executed by evaluating the position list P and condition D on C. Our system allows for two execution strategies:

i ship all partitions $C_1, \ldots, C_n$ from nodes $RN_1, \ldots, RN_n$ to AN and evaluate P and D locally at AN. We denote this strategy *data pull* (DP).

ii ship P and D to RAMCloud nodes and evaluate them remotely at $RN_1, \ldots, RN_n$. Here, P has to be split-up into sub-partition lists $P_1, \ldots, P_n$ to ship a specific position lists to each RAMCloud node. We denote this strategy by *operator push-down* (OP).

To push down an operation O to RAMCloud nodes, it is split-up into sub-operations $O_1, \ldots, O_n$. These sub-operations O_i take a condition D and a specific position list P_i as input to be evaluated on all values in C_i. Since we measure the network traffic in bytes we have to distinguish different cases for each operator: for a scan operation, D is the selection condition and usually only a few bytes large (e.g. the size of two scan comparators and two comparative values). In case of a materialization operation we set $D = \emptyset$ to return all values defined in P. For a join operation, D denotes the probing data and has a significant size. We denote the size of D in bytes by S_D. The output of O_i is a list of column values or column positions where D evaluates to true. In our model the fraction of values referenced in P_i for which D evaluates to true is defined by the selectivity parameter s. In case $P_i = \emptyset$, then D is applied to all values at C_i. We denote the number of entries in P by S_P and the size of one entry in bytes by S_p.

5.2 Execution Cost

To derive the overall time E_O required to execute an operation O for DP and OP analytically, we first derive the delay induced by network transfers and afterwards the times required to execute operators in local DRAM.

For operation O applied to a column C partitioned over n RAMCloud nodes, we derive network costs M as follows: For DP the network cost are simply given by

$$M_{DP} = S_C \cdot S_r / BW_{Net} \tag{1}$$

because their only dependency is the amount of data that is pulled from $RN_1, \ldots, RN_n$ to the local execution on AN.

For OP network costs M_{OP} depend on the size of D and P, as well as the selectivity of the predicate s. We derive M_{OP} in (2):

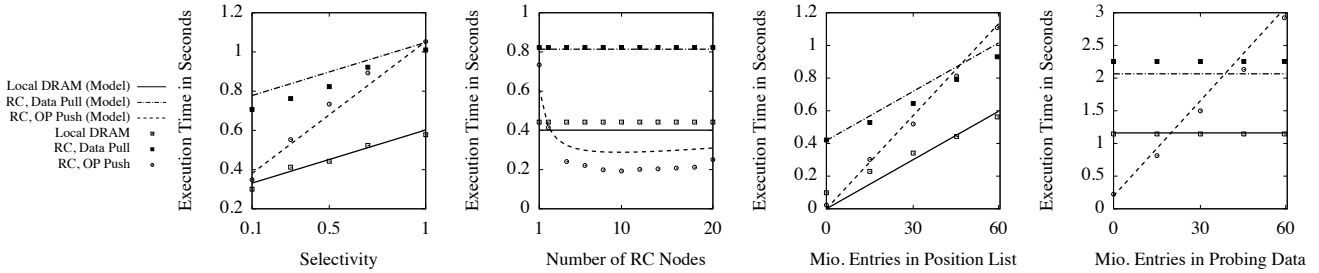
$$M_{OP} = ((S_P \cdot S_r + S_D \cdot n) + (S_P \cdot s \cdot S_r))/BW_{Net} \tag{2}$$

The time required to execute a O_i on a RAMCloud node RN_i in case of OP depends on the scan speed in DRAM at RN_i, and in case of DP on the scan speed at AN. In our system model we define the scan speed by parameter BW_{Mem}. We abstract the execution time of an operation T_O as the sum of the time required to traverse the data and the time to write results as follows:

$$T_O = (S_P \cdot S_r + S_P \cdot S_r \cdot s)/BW_{Mem} \tag{3}$$

If the operation is a sub-operation O_i the execution time $T_{O,i}$ at RN_i is derived similar to (3) by using node specific position list sizes $S_{P,i}$ instead of S_P.

In case of OP we have to consider the overhead time T_{ovh} required to split P and later merge results received from RAMCloud



(a) Scan with increasing selectivity. S_D=2, S_P=0, s={0.1...1}, n=1, BW_{Mem}=2GB/s

(b) Scan with increasing number of nodes. S_D=2, S_P=0, s=0.5, n={1...20}, BW_{Mem}=2GB/s

(c) Materialization with increasing position list size. S_D=0, S_P={0...60 Mio.}, s=1, n=1, BW_{Mem}=2GB/s

(d) Join Probing with increasing probing data size. S_d={0...60 Mio.}, S_P=10 Mio., s=0.5, N=20, BW_{Mem}=0.4GB/s

Figure 4: Modeling and validating the execution times for different storage options and operator execution strategies with the following fixed parameters: S_C=60 Mio., S_r=S_p=8bytes, BW_{Net}=2.2GB/s. The diagrams show how the variation of the parameters impacts the execution time for the different operator execution strategies. I.a. Figure 4(c) and 4(d) illustrate that a data pull execution can become preferable to an operator push execution.

nodes $RN_1, \ldots, RN_n$. This results in an in-memory traversal over the operators input and out put data. We define T_{ovh} as

$$T_{ovh} = S_P \cdot n/BW_{Mem}/S_p + (S_P \cdot s \cdot S_r)/BW_{Mem} \quad (4)$$

We now derive the overall execution time E_O in seconds of operation O by the sum of required network transfer time M and the time for operator execution, distribution and merge overhead. For DP we derive (5).

$$E_{O,DP} = M_{DP} + T_O \quad (5)$$

For OP we have to consider the overhead for computing n specific position lists as well as the merge of O_i results. Hence, we derive (6) as execution time for OP.

$$E_{O,OP} = M_{OP} + T_{ovh} + max(T_{O,i}) \quad (6)$$

While our cost model abstract from numerous system parameters, we found this abstraction accurate enough to evaluate the impact of our operator execution parameters.

5.3 Evaluating Operator Execution Strategies

After introducing the AnalyticsDB execution cost model, we validate it with a set of micro benchmarks. Each micro benchmark represents a single operator execution. We vary the previously described cost model parameters throughout the micro benchmarks, so that they allow for a discussion about the relation of the execution strategies data shipping and operator push down. We execute the micro benchmarks on a cluster of 50 nodes in total where each node has an Intel Xeon X3470 CPU, 24GB DDR3 DRAM, and a Mellanox ConnectX-2 InfiniBand HCA network interface card. The nodes are connected via a 36-port Mellanox InfiniScale IV (4X QDR) switch. We use one node for running AnalyticsDB and vary the number of RAMCloud nodes between one and 20. The chosen number of nodes is sufficient for demonstrating the impact of the cost model parameters on the operator execution, we will use the full cluster capacity in the subsequent Performance Evaluation Section. In addition, we provide a baseline where the respective micro benchmark is executed on local DRAM on a single AnalyticsDB node.

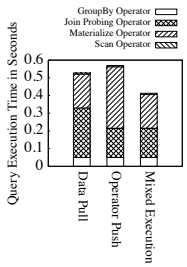
Figure 4 depicts our micro benchmarks. There are three parameters which are fixed throughout all benchmarks: the column size S_C=60 Mio. and the size of a single column record S_r=8 bytes. The effective network bandwidth is BW_{Net}=2.2GB/s: the theoretical maximum network bandwidth is 4GB/s with the particular InfiniBand hardware, but it is limited by the PCI Express bandwidth in the nodes.

Figure 4(a) shows a scan operation on the entire column with an increasing selectivity and a fixed RAMCloud cluster size of one. The operator push execution time for a scan with a low selectivity is close to the local DRAM variant, data pull takes almost twice as long due to the initial full column copy over network. With an increasing selectivity, the execution time of the operator push down strategy approaches the data pull variant as the same amount of data travels over the network.

Figure 4(b) illustrates a full column scan with a fixed selectivity of 0.5, but with a varying number of nodes in the RAMCloud cluster. One can see that with an increasing number of nodes, the operator push down execution gets accelerated due to the parallel execution of the scan operator, but reaches a limit at around 10 nodes: at that point $T_{O,i}$ is minimized and the execution time is dominated by $M_{OP} + T_{ovh}$ which cannot be reduced by adding more nodes. T_{ovh} even grows with an increasing number of nodes which causes the operator execution time to slightly increase towards a cluster size of 20 nodes. The execution time of data pull is not affected by a larger cluster size and is constant.

Figure 4(c) depicts a materialization operation with an increasing position list size. The operator push down execution time increases gradually with a growing position list size and exceeds at some point the execution time of data pull. This is caused by the addition of the size of the position list and the returned column values which exceeds the column size. In such a case, the data pull execution time is faster than the operator push execution time.

Figure 4(d) shows a join probing with an increasing probing data size on a RAMCloud cluster with 20 nodes. BW_{Mem} is here smaller than in the previous micro benchmarks since the creation of and the probing against a hash map takes longer than a scan or a materialization operation. The graph illustrates that operator push down can benefit from parallel execution on 20 nodes and is faster than execution on local DRAM when the probing data is small. When the probing data gets bigger, the operator push down execution time increases as the probing data has to be sent to all 20 nodes. At a certain probing data size, the operator push down execution time exceeds the data pull execution time and makes data



(a)

```
select sum(lo_revenue)
from lineorder, part
where lo_partkey = p_partkey
and p_mfgr between 'MFGR#1'
and 'MFGR#5';
```

(b)

Figure 5: Execution of a SQL query on the SSB dataset with SF=1 (S_C=6 Mio., S_r=8bytes), N=10 and different execution strategies. Figure 5(a) illustrates that for this query a mix of data pull and operator push execution strategies is preferable.

pull preferable.

Summarizing the gained insights based on the micro benchmarks, the data pull execution strategy is two to three times slower than operating on local DRAM. The performance of the operator push strategy varies: if the to be accessed data is on a single RAMCloud node, the performance is only a few percent worse than operating on local DRAM if e.g. the selectivity or the number of entries in the position list is small. If those parameter grow, the operator push performance gradually approximates the data pull performance and can even become worse. If the to be processed data is partitioned across several nodes, the operator push execution time can be up to five times faster than local DRAM. But node parallelism can also worsen the operator push execution time to an extent that it becomes slower than data pull: this is the case if the input parameter S_d becomes large and must be dispatched to all involved nodes.

5.4 Optimizing Operator Execution

The previous subsection demonstrated that the optimal operator execution strategy depends on a set of parameters. In this subsection, we show that the optimal execution strategy within a single query can vary for each involved operator. We use the same cluster setup as in the previous subsection.

The Figure 5(a) depicts the execution times for different execution strategies based on the query shown in 5(b). The join probing operation (S_d=200.000, S_P=6 Mio., s=1) can benefit from the parallelism of the ten nodes and is fastest with a operator push strategy. The materialization operation (S_d=0, S_P=6 Mio., s=1) has a position list size that is as large as the column itself: the data pull strategy performs better for this operator execution. Consequently, the optimal execution time can be reached with a mix of the data pull and operator push strategies as illustrated by the last column in Figure 5(a).

6. PERFORMANCE EVALUATION

In this section we present a detailed performance evaluation by executing the Star Schema Benchmark that has been described in Subsection 2.1. We describe how the execution time is impacted by the different operator execution strategies, by an increasing data scale factor, and by the different data partitioning options. In addition, we inspect the elasticity of the presented architecture by changing the number of RAMCloud nodes, changing the number of AnalyticsDB nodes and combining both aspects by showing how to maintain a constant response time under a variable load. We use the same cluster setup as described in Subsection 5.3. Since RAMCloud supports currently only a single-threaded operator execution per node, we also execute the queries in the AnalyticsDB nodes single-threaded.

Figure 6 shows an AnalyticsDB operator breakdown for each query of the SSB. Each query is executed on local DRAM and on RAMCloud. AnalyticsDB runs on a single node, the RAMCloud cluster has size of 20 nodes. The execution on RAMCloud happens either via data pull or operator push strategy and each AnalyticsDB column is either being stored on one storage node (server span=1) or partitioned across all nodes (server span=20). The figure illustrates that the partitioning criteria has only very little impact (2.8%) on the data pull execution strategy and that data pull is in average 2.6 times slower than the execution on local DRAM. With a server span of one, the operator push execution strategy is in average 11% slower than the execution on local DRAM. With a server span of 20, the operator execution strategy can be accelerated by a factor of 3.4: the next subsection discusses the impact of data partitioning on the SSB execution time in detail. The figure does not show the execution times of the local AnalyticsDB operators such as Sort in detail, but summarizes them as *Other Operators*. Due to the overall low selectivity of the SSB, there is no case where a Scan, Materialize, or Join Probing Operator execution is slower with an operator push than with a data pull execution strategy.

6.1 Data Partitioning

As seen in the previous subsection, the partitioning criteria influences the execution time of the SSB with an operator push execution strategy. As explained in Subsection 5.3, an operator push execution can benefit from a parallel execution on several nodes until the execution time of the operation time is minimized and the execution time is dominated by the data transfer over network and overhead costs such as merging the results from all nodes. The micro benchmark in Figure 4(b) illustrated this with a column size of S_C=60 Mio. values. The SBB data set has different tables with different column lengths. Figure 7 shows the combined execution times of all scan operations on the SSB Lineorder, Part, and Date tables during a single SSB cycle. At a SSB data scale factor of 10, each column in the Lineorder table has a size of S_C=60 Mio. values, each column in the Part Table has a size of S_C=800.000 values, and each column in the Date table has a size of S_C=2.556 values. Figure 7 depicts that the scan operations on the Lineorder table benefit up to a factor 5 from being distributed, the scan operations on the Part table get accelerated up to factor 1.6, but the scan operations performance on the Date table decreases with every additional node up to a factor 4.5. This raises the question if the chosen partition criteria should not be derived from the optimal partitioning layout for the biggest table, but being done independently for each table or column, depending on the respective column length: we disregard this idea as the introduction of a column specific partitioning a) makes the data migration during a scale out more complex (as discussed in the next Subsection 6.3) and b) only brings a comparatively small performance benefit (e.g. 3.8% in the example in Figure 7). In addition, we have so far only covered the aspect of data partitioning when one AnalyticsDB instance operates exclusively on a RAMCloud cluster. We will cover the aspect of multiple AnalyticsDB instances in the upcoming Subsection 6.4.

6.2 Data Scale Factor

This subsection evaluates what impact a varying data set size has on the execution time. Figure 8 shows the execution of the SSB with a varying data scale factor SF. Scale factor 1 has a Lineorder table with 6 million rows and a total data size of 600 MB, scale

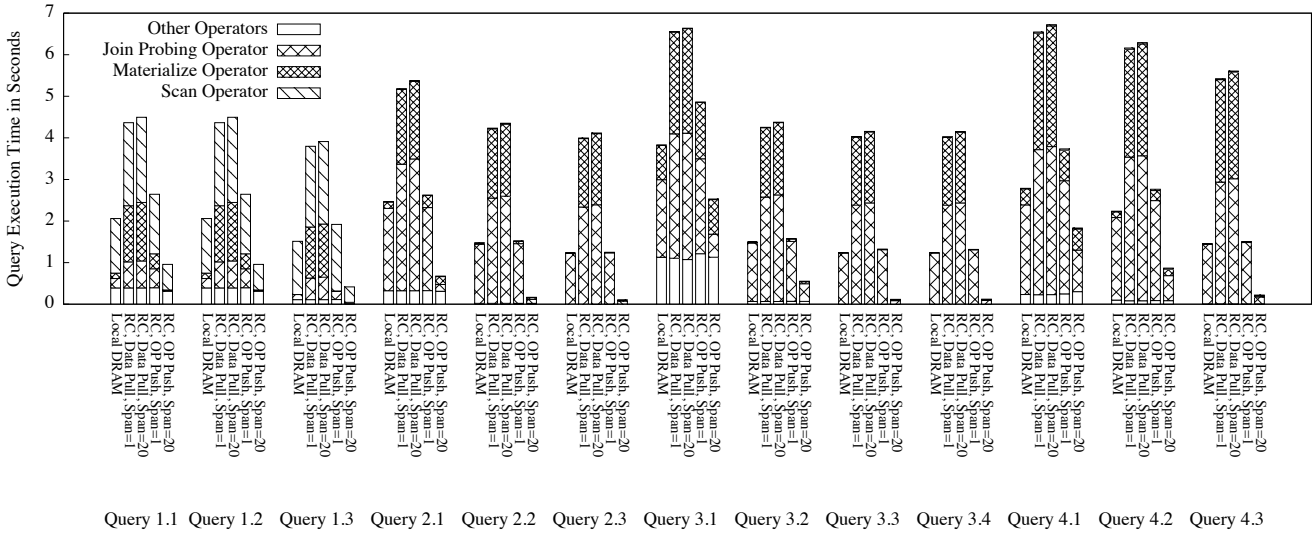


Figure 6: Operator breakdown for AnalyticsDB executing SSB queries with a data scale factor of 10 and different storage options and operator execution strategies. AnalyticsDB runs on a single node, the RAMCloud cluster has size of 20 nodes. The figure illustrates i.a. that the data pull execution strategy is in average 2.6 times (or 260%) slower than the execution on local DRAM and that the operator push execution strategy is in average 11% slower than the execution on local DRAM.

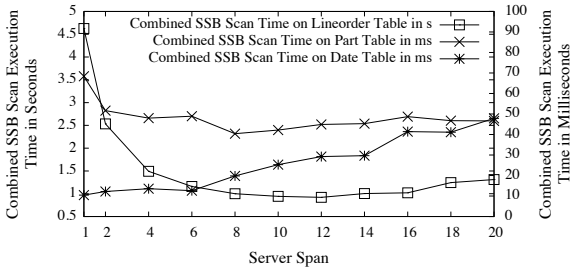


Figure 7: Combined scan operation times on SSB tables with a data scale factor of 10. Analytics DB runs on a single node with a operator push execution strategy, the RAMCloud cluster has a size of 20 nodes, the server span varies.

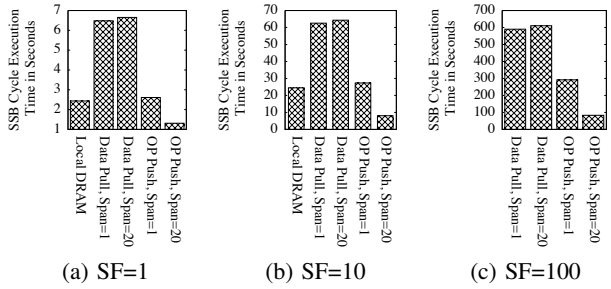


Figure 8: RAMCloud cluster with 20 nodes and a single node running AnalyticsDB with a varying SSB data scale factor SF. The figure shows that the ratio between data set size and SSB execution times remain constant with a growing data set size.

factor 10 has a Lineorder table with 60 million rows and a total data set size of 6 GB, and scale factor 100 has 600 million rows in the Lineorder table and a total data set size of 60 GB. The experiments with SF 100 could not be executed on local DRAM as the data set size exceeded the capacity of a single server. Figure 8 illustrates that the ratio between the data set size and the SSB execution times of the different execution strategies remain constant with a growing data set size and with a constant cluster size.

6.3 Elasticity: Variable Number of RAMCloud Nodes

Throughout previous experiments, we varied the number of nodes in the RAMCloud cluster and the resulting server span. In this subsection, we want to perform this variation not in separate experiment executions, but continuously while a single AnalyticsDB node is executing queries. Therefore, we use a simplistic data migration manager which distributes the data equally across the available nodes: if a new node joins the RAMCloud cluster, it gets a chunk of the data, before a node is removed from the cluster its contained data is distributed across the remaining nodes. The data distribution is done via a splitting of the RAMCloud namespaces (see Section 3.1) and a subsequent migration of the data that is contained in a part of a namespace: the complexity and execution time of this mechanism benefits from an equal partitioning of all namespaces.

Figure 9 illustrates the SSB execution time while RAMCloud nodes are being added or removed from the cluster. With every added RAMCloud node, the overall storage capacity increases and the SSB execution time decreases as previously discussed. With every removed node the overall storage capacity decreases and the SSB execution time increases.

6.4 Elasticity: Variable Number of AnalyticsDB Nodes

In this subsection, we have a constant number of 20 nodes in the RAMCloud cluster, but vary the number of nodes that execute AnalyticsDB between 1 and 30. If a new AnalyticsDB node is

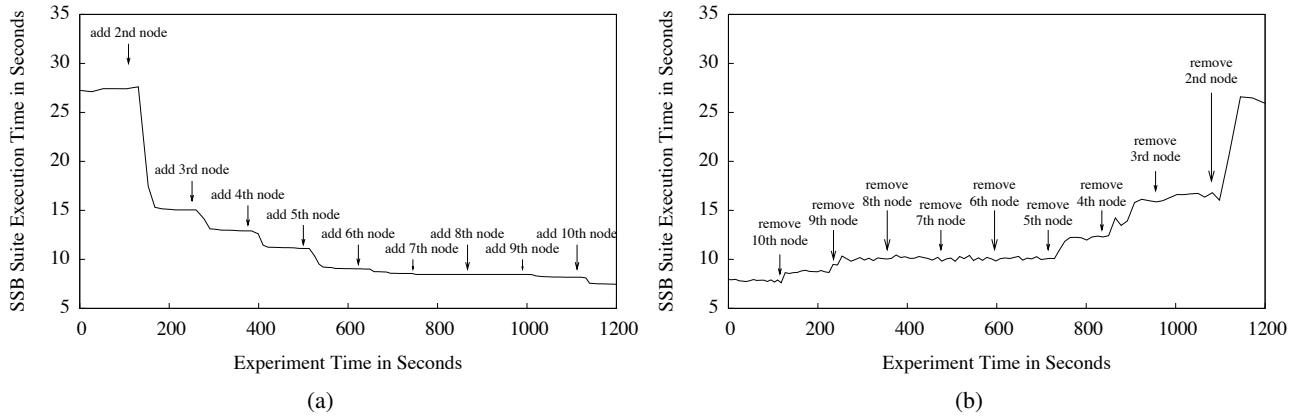
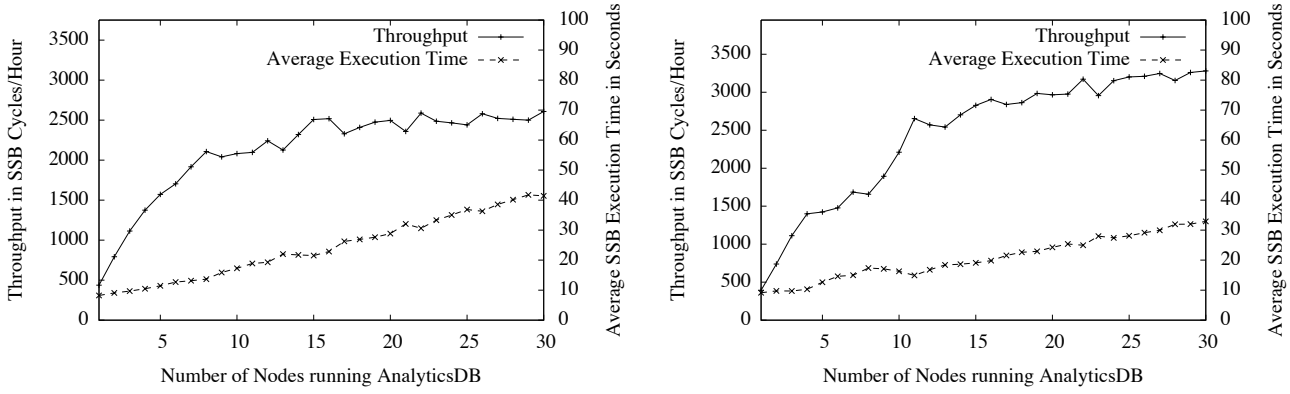


Figure 9: RAMCloud cluster with a varying number of nodes and a single node running AnalyticsDB with a operator push execution strategy and a SSB data scale factor of 10.



(a) RAMCloud running on 20 Nodes with Server Span=10 (b) RAMCloud running on 20 Nodes with Server Span=20

Figure 10: RAMCloud cluster with a constant number of 20 nodes and a varying number (1-30) of nodes running AnalyticsDB with a operator push execution strategy and a SSB data scale factor of 10.

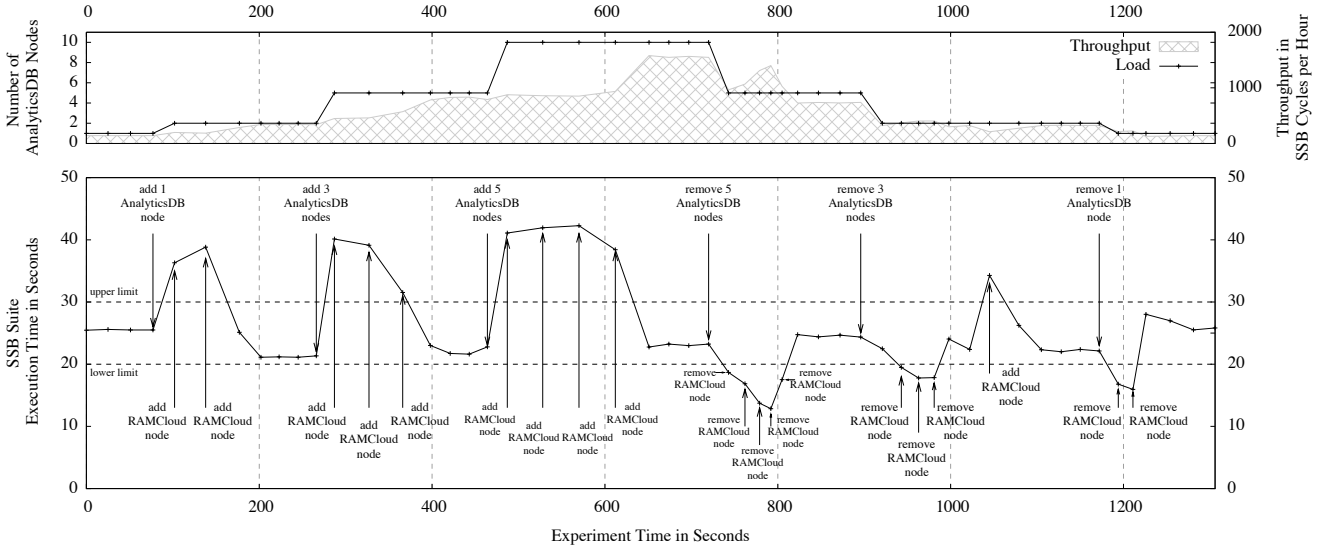


Figure 11: RAMCloud cluster with a varying number of nodes that react on workload changes which are imposed by a changing number of AnalyticsDB nodes. The SSB data scale factor is set to 10, the execution strategy is operator push. The figure illustrates i.a. that our architecture can utilize resources which have been added due to an increased workload after a short period of time.

added, it gets instructed by the federator to continuously execute the SSB: this results in an increase of the load. Figure 10 shows the corresponding experiment, where in Figure 10(a) the server span is 10 and in Figure 10(b) the server span is 20. In Figure 10(a) the throughput increases until 15 AnalyticsDB nodes and then begins to flatten out which means the operator throughput is saturated in RAMCloud. The maximum throughput is 2607 SSB cycles per hour. In Figure 10(b) the throughput increases until 20 AnalyticsDB nodes and then begins to flatten out. The maximum throughout is 3280 SSB cycles per hour.

The following two insights can be derived from the experiments: a) we demonstrated in Subsection 6.1 that a server span of 10 delivers the optimal SSB execution time when a single AnalyticsDB node uses a RAMCloud cluster with 20 nodes. This statement is valid if there are up to ten AnalyticsDB nodes running. Above ten AnalyticsDB nodes a server span of 20 results in a better SSB execution time as the to be accessed data is distributed across more RAMCloud nodes and therefore the operator throughput in RAMCloud is saturated at a later point. b) Even in the case of overprovisioning (e.g. 30 AnalyticsDB nodes vs. 20 RAMCloud nodes) the SSB throughput remains constant, but the execution time increases over linear (due to the operator throughput saturation in RAMCloud), but it does not result e.g. in a reduction of the SSB throughput. In addition, the increasing throughput in both experiments can either be leveraged for performing a higher number of SSB executions in parallel or for reducing the execution time of a single SSB execution by dispatching its queries across the different AnalyticsDB nodes via the federator.

6.5 Elasticity: Constant Execution Time under a Variable Load

In the previous two subsections, we varied either the number of RAMCloud nodes or we varied the load by changing the number of AnalyticsDB nodes. In this section we want to put the pieces together by maintaining a constant SSB execution time by resizing the RAMCloud cluster online under a changing load which is represented by a varying amount of AnalyticsDB nodes.

Figure 11 shows that we vary the load in the experiment by adding and removing a set of one, three, and five AnalyticsDB nodes at a time over the course of the experiment. Every time an AnalyticsDB node has been added, it is told by the federator to constantly execute the SSB. This experiment defines an upper and lower execution time limit for the average SSB execution time of 30 and 20 seconds. If the load has been increased by adding AnalyticsDB nodes, then a new RAMCloud node is being added to the cluster for every execution time measuring point that is above the upper limit. The same approach is used when AnalyticsDB nodes are being removed and an execution time below the lower limit results in the removal of RAMCloud nodes. Although the chosen work load adaption strategy is most simplistic, the experiment shows that a) the architecture can adapt to workload changes of different orders in a short period of time, b) without interrupting the ongoing query processing and c) that the resulting elasticity allows the compliance with a performance goal without any adjustments from a DBMS perspective.

7. RELATED WORK

The most important related work is from Brantner et al. who demonstrated [4] that a cloud-based storage system can be used as a shared-storage for a database application by putting MySQL on Amazon S3. Our paper has the same intent, but differs as it focuses on performance and elasticity in the context of an analytical workload and a DRAM-based storage system.

The concepts which influence and made up the different pieces of our architecture are covered in the distributed systems and database literature:

- The shared-nothing vs. shared-disk (shared-storage) discussion has been extensively covered e.g. by Wong and Katz [28], Stonebraker [23] and Rahm [20].

- The aspect of bringing the executing of operations to the data storage is well established in the field of database and distributed systems: in the context of database systems, database-aware storage systems enable the push-down of database operators into the storage system as shown by Sivathanu et. al. [22] and Raghuveer et. al. [19]. In the context of distributed systems, e.g. the Hadoop Distributed File System [3] provides interfaces for applications to move their execution closer to the data.

- There are several approaches for combining the main memories of several machines for a centralized application: the concept of distributed shared memory [13] does this by providing a virtual memory address space and research projects such as MEMSCALE [12] adapt this concept to the properties of modern hardware. However, such an approach is not intended for a large number of servers with frequent hardware failures, but is rather suited for a small set of highly reliable servers. In contrast, DRAM-based storage systems such as RamSan [26] provide high-availability, but are limited in their maximum storage capacity and do not allow the invocation of remote operations.

In addition, there is an on-going discussion whether the MapReduce framework [6] or distributed, parallel DBMSs [24] are the right tool for performing analytics on large data sets: the spectrum of this discussion also includes hybrid approaches such as Google's Dremel [11]. Our work is positioned in the realm of distributed DBMSs and utilizes a cloud infrastructure as storage system: although the execution of a set of database operators is being brought closer to the data into the cloud-based storage, the data model, the query syntax, and the query execution is designed for and controlled by the DBMS.

8. CONCLUSION

This work presents AnalyticsDB, a system that combines the performance of an in-memory query processor and the elasticity of a cloud data storage. The technological enabler for this combination is modern computer networking technology. The conceptual enabler is the DBMS architecture presented in this paper.

From a performance point of view, we show that a) using a vanilla RAMCloud instead of local DRAM for data storage results in a performance penalty of a factor 2.6, but enables already all the advantages of RAMCloud such as fault-tolerance, availability, and elasticity. With the enrichment of RAMCloud by a set of data-intensive operators we demonstrate that b) the performance penalty can be reduced to 11%. From an elasticity perspective, we illustrate that c) our architecture can adapt to a changing number of storage nodes with no efforts from a DBMS perspective, without an interruption of the query processing, and within a few seconds. In addition, we point out that this is also d) true for a changing number of DBMS nodes. Combining both previous statements results in the illustration of e) the ability to meet a certain performance goal under a changing workload by adding/removing resources. The conducted experiments use a widely known analytical benchmark

with a data size of up to 600 million records and a cluster size of up to 50 nodes.

Throughout the description of aspects which are unique to our system we introduce f) a query execution cost model that allows to decide whether a data pull or an operator push execution strategy is preferable for a given query. Besides optimizing the query execution, we also evaluate the implications on finding an optimal partitioning schema. We show that g) the optimal partitioning schema varies from a single DBMS node and from an overall cluster perspective.

Putting all pieces together, our architecture and its API between query execution engine and data access can use a large-scale DRAM-based storage system such as RAMCloud as data storage instead of local DRAM. This enables the preservation of the in-memory performance advantage and the elasticity provided by the cloud storage at the same time. From our point of view, the closing gap between local and remote DRAM access performance characteristics will cause a reevaluation of established concepts and common knowledge in the field of distributed in-memory DBMSs. In this work, we made a first step towards leveraging the relaxed remote performance constraints for providing a greater degree of elasticity.

9. REFERENCES

[1] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. Madden. Materialization Strategies in a Column-Oriented DBMS. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 466–475. IEEE, 2007.

[2] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51(12):77–85, 2008.

[3] D. Borthakur. *The Hadoop Distributed File System: Architecture and Design*. The Apache Software Foundation, 2007.

[4] M. Brantner, D. Florescu, D. A. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 251–264. ACM, 2008.

[5] C. Curino, E. P. Jones, S. Madden, and H. Balakrishnan. Workload-aware database monitoring and consolidation. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 313–324, New York, NY, USA, 2011. ACM.

[6] J. Dean and S. Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, Jan. 2010.

[7] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[8] InfiniBand Trade Association. The InfiniBand Architecture.

[9] Intel Coporation. Intel Xeon Processor E5-4650 Specification.

[10] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, Dec. 2000.

[11] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3(1-2):330–339, Sept. 2010.

[12] H. Montaner, F. Silla, H. Fröning, and J. Duato. Memscale: in-cluster-memory databases. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, CIKM '11, pages 2569–2572, New York, NY, USA, 2011. ACM.

[13] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, Aug. 1991.

[14] P. E. O'Neil, E. J. O'Neil, X. Chen, and S. Revilak. The star schema benchmark and augmented fact table indexing. In R. O. Nambiar and M. Poess, editors, *TPCTC*, volume 5895 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2009.

[15] O'Neil, P. E. and O'Neil, E. J. and Chen, X. The Star Schema Benchmark (SSB).

[16] D. Ongaro, S. M. Rumble, R. Stutsman, J. K. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 29–41. ACM, 2011.

[17] J. K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. M. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMCloud. *Commun. ACM*, 54(7):121–130, 2011.

[18] M. T. Ozsu. *Principles of Distributed Database Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[19] A. Raghuveer, S. W. Schlosser, and S. Iren. Enabling database-aware storage with osd. In *MSST*, pages 129–142. IEEE Computer Society, 2007.

[20] E. Rahm. Parallel query processing in shared disk database systems. *SIGMOD Rec.*, 22(4):32–37, Dec. 1993.

[21] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout. It's time for low latency. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, HotOS'13, pages 11–11, Berkeley, CA, USA, 2011. USENIX Association.

[22] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-dusseau, and R. H. Arpaci-dusseau. Database-aware semantically-smart storage. In *In Proceedings of the 4th USENIX Conference on File and Storage Technologies. USENIX Association*, pages 239–252, 2005.

[23] M. Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[24] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel dbmss: friends or foes? *Commun. ACM*, 53(1):64–71, Jan. 2010.

[25] M. Stonebraker, C. Bear, U. Çetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. B. Zdonik. One Size Fits All? Part 2: Benchmarking Studies. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 173–184. www.cidrdb.org, 2007.

[26] Texas Memory Systems. TMS RamSan-440 Details.

[27] C. Tinnefeld, A. Zeier, and H. Plattner. Cache-conscious data placement in an in-memory key-value store. In *15th International Database Engineering and Applications Symposium (IDEAS 2011), September 21 - 27, 2011, Lisbon, Portugal*, pages 134–142. ACM, 2011.

[28] E. Wong and R. H. Katz. Distributing a database for parallelism. *SIGMOD Rec.*, 13(4):23–29, May 1983.