On-Demand State Separation for Cloud Data Warehousing

Christian Winter
Technical University of Munich
winterch@in.tum.de

Jana Giceva
Technical University of Munich
giceva@in.tum.de

Thomas Neumann
Technical University of Munich
neumann@in.tum.de

Alfons Kemper
Technical University of Munich
kemper@in.tum.de

ABSTRACT

Moving data analysis and processing to the cloud is no longer reserved for a few companies with petabytes of data. Instead, the flexibility of on-demand resources is attracting an increasing number of customers with small to medium-sized workloads. These workloads do not occupy entire clusters but can run on single worker machines. However, picking the right worker for the job is challenging. Abstracting from worker machines, e.g., using stateless architectures, introduces overheads impacting performance. Solutions without stateless architectures resort to query restarts in the event of an adverse worker matching, wasting already achieved progress.

In this paper, we propose migrating queries between workers by introducing on-demand state separation. Using state separation only when required enables maximum flexibility and performance while keeping already achieved progress. To derive the requirements for state separation, we first analyze the query state of medium-sized workloads on the example of TPC-DS SF100. Using this, we analyze the cost and describe the constraints necessary for state separation on such a workload. Furthermore, we describe the design and implementation of on-demand state separation in a compiling database system. Finally, using this implementation, we show the feasibility of our approach on TPC-DS and give a detailed analysis of the cost of query migration and state separation.

1 INTRODUCTION

The high flexibility and cost-efficiency of cloud databases, such as Snowflake [12] and Amazon Redshift [24], are attracting an increasing range of customers. While these systems offer solutions for petabytes of data and optimize for scalability, they also attract
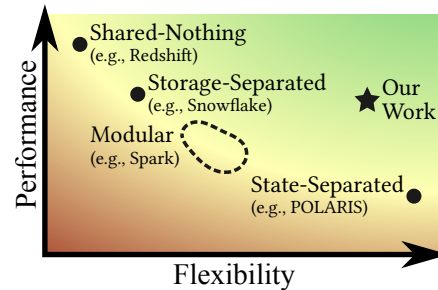
Figure 1: Classification of cloud data warehouse architectures by performance and flexibility. Flexibility here is the ability to adapt to changes in the execution environment and provide scalability for processing. Performance is defined by query throughput and latency.

smaller customers and workloads. These workloads do not require the full elasticity offered and can often be handled by one or a few machines. However, finding the optimal instance to provide cost optimality is still not trivial [35]. To understand the challenges of cloud data warehousing for smaller workloads, one needs to look at the dominant warehouse architectures and their characteristics outlined in Figure 1.

First, there is the classic shared-nothing architecture prominent in on-premise deployments and used in Amazon Redshift [24]. In this architecture, both storage and compute are co-located on a worker. While this offers the best performance, it cannot scale resources independently. Second, there are storage separated architectures, such as Snowflake [12]. These allow compute and storage to be scaled separately. Keeping the working state of a query at the compute node still achieves excellent performance. However, this does not permit elasticity and fault tolerance for individual queries. Third, state-separated architectures, like Microsoft POLARIS [2], fully decouple state and compute. The high flexibility and elasticity of state separation come at the cost of network overhead when syncing the state between tasks. This overhead is acceptable when data has to be shuffled between workers after each task. Finally, modular systems, like Apache Spark [65], do not follow a specific architecture fully but can be configured similarly to one or more architectures. For Spark, e.g., state separation can be achieved by strategically placing checkpoints in the query plan [44, 60]. We argue that, due to network transfer costs, stateless architectures are not profitable for smaller workloads. Nevertheless, there is a growing need for higher flexibility for such workloads: Ambati

```
with customer_revenue as (
  select customer.id, sum(orders.price) as revenue
  from customer, orders
  where customer.id = orders.customer_id
  group by customer.id
)

select c.name, c.adress, c.birthday
from customer c, customer_revenue r
where c.id = r.id and
  r.revenue >= 0.9 * (
    select max(revenue) from customer_revenue
  )
```
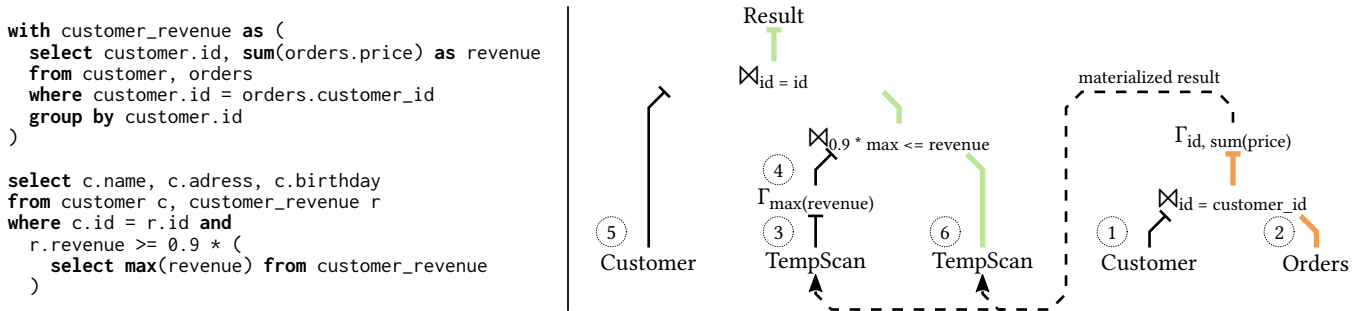

Figure 2: Exemplary SQL OLAP query (left) and corresponding query plan (right) with pipelines. Pipelines continuing through an n-ary operator are marked bold and color-coded. Circled pipeline IDs also denote execution order.

et al. [5] propose speculatively executing queries to find the best query-to-worker matching, losing all progress achieved when the worker has to be changed. In addition, Garefalakis et al. [21] have described the need for suspendable tasks to provide low latency for time-sensitive tasks when resources are limited.

In this paper, we propose on-demand state separation to provide the desired flexibility of stateless architectures without incurring the performance cost. This way, we can utilize the full performance of storage-separated architectures for local queries, while still allowing for query migration and elasticity with minimal overhead when necessary. To achieve this, we run the query as if it were only storage-separated. When the need for state separation arises, we cache all relevant intermediate results over the network and continue the query on these on the target worker. This query migration can be beneficial in a number of settings. It allows the utilization of more powerful or cheaper servers that become available in a cluster for already running queries. Furthermore, it enables load balancing between servers in multi-tenancy settings, as well as the utilization of spot instances for query processing. Queries can be started on such instances and migrated when the spot instance expires, which has been proposed for VM instances [45, 48, 61] and using predefined checkpoints [44, 60] in Apache Spark. Contrary to those solutions, we only need to migrate the current working state of a query and do not require a priori knowledge about the workload. Our key contributions are:

- We provide an analysis of the query states occurring in mid-sized cloud workloads on the exemplary workload of all queries of the TPC-DS benchmark [52] at a scale factor of 100. This dataset of roughly 100GB represents a medium-sized workload, which can be reasonably executed on a single server. To the best of our knowledge, we are the first to describe query states in a state-separated architecture.
- We describe the constraints for the deployment environment necessary for (on-demand) state-separated architectures on such workloads.
- We show the design and implementation of on-demand state separation in an OLAP database system using the code-generating DBMS Umbra [40].
- We evaluate the performance and overhead of on-demand state separation for various use cases.

The remainder of the paper is structured as follows: Section 2 defines the goal of on-demand state separation and the relevant concepts. Then, we analyze the state of OLAP workloads based on TPC-DS in Section 3. Section 4 describes the design and implementation of on-demand state separation, which we evaluate in Section 5. We discuss related work in Section 6 before concluding in Section 7.

2 PROBLEM DEFINITION

Before describing our novel approach to on-demand state separation, we first need to formalize the problem statement, as well as the requirements that queries and systems have to fulfill to support our approach. The goal of on-demand state separation is to provide flexibility for traditional relational databases in cloud settings. To achieve this flexibility, we utilize state separation [2].

DEFINITION 1. **State Separation:** *State separation is the process of decoupling the working state and progress of a query from the machine executing it.*

A state-separated query can, thus, be resumed on any machine, even if the new machine's configuration differs from the old one. Changing the executing server at runtime has been employed in the past, e.g., in multi-engine environments [1, 46]. These systems focus on migration between different engines at pre-planned points in the query plan for performance. In contrast, we aim for flexibility by migrating on demand without prior planning.

GOAL. *On-demand state separation achieves state separation retroactively with minimized progress loss and minimized query state without hampering the performance of local execution.*

2.1 Background

Having defined the goal of on-demand state separation, we need to discuss the requirements a database system has to fulfill to support it, as well as the properties of relational queries we use for our approach. For this, we will use the exemplifying query in Figure 2 and its query plan, which serves as a running example throughout this paper. First, we find high-grossing customers using the common table expression *customer_revenue*. Then, we reconnect these customers to the customer table to extract all information required to send them birthday cards.

2.1.1 Query Properties. Our approach is based on relational queries following a query plan such as the one displayed in Figure 2. Plans

consist of operators, such as joins, aggregations, and filters. These operators can be grouped into two categories, blocking and filtering operators. Blocking operators, such as the *id, sum* aggregation, materialize all tuples before reporting the result. Filtering operators, on the other hand, do not materialize tuples. Operators can exist as a filtering and a blocking operator simultaneously. The *id = customer_id* join, e.g., will materialize all tuples from the customer relation but only filter tuples from the orders relation without materializing them. We call all paths in the query plan in which a tuple is not materialized, i.e., between two blocking operators, pipelines.

Each pipeline is executed exactly once for all tuples of its input. Nevertheless, the result of a pipeline may be used multiple times. The result of pipeline (2) in Figure 2, for example, is scanned twice, namely in pipelines (3) and (6). Furthermore, a pipeline fully depends on all of its inputs. Before pipeline (6) can start, pipelines (2), (4), and (5) must finish their execution. These dependencies result in the execution order denoted by the pipeline IDs in Figure 2.

2.1.2 System Requirements. A system has to fulfill three main requirements to support on-demand state separation, which we will discuss below. As we have implemented our approach within the Umbra database system [29, 40, 57, 58], we give a brief overview of how Umbra adheres to these requirements.

Plan-Based Execution. In our approach, we process queries using relational operators and pipelines. Therefore, we also require the system to process queries based on relational operators. Query plans are the default execution model for relational databases. Therefore, most existing database systems adhere to this requirement. Furthermore, the system must support the serialization and deserialization of plans for execution, either through dedicated formats or by emitting SQL. Umbra, e.g., uses a pipeline-based execution model. In it, the query is split into pipelines, which in turn are translated into code and compiled for execution. Umbra further supports the export and import of query plans to and from JSON format, which we use to share queries between instances.

Query Progress Information. As one goal is to preserve already achieved query progress, systems have to offer insights into the progress of running queries. This progress information is already part of query execution in interpreting systems, such as MonetDB [10]. Compiling systems, such as Umbra, which convert queries to machine code, require active progress-keeping. In Umbra, it is not the entire query but its individual pipelines that are converted to machine code. This pipeline-based conversion allows us to keep track of the query progress at pipeline granularity.

Accessible Intermediate Results. Finally, our approach requires access to the intermediate results held for a query as they materialize the progress achieved. While interpreting engines access these results directly for query processing, compiling engines typically only access them through generated code. For state separation, however, compiling systems need to maintain information on these intermediate results outside of generated code as well. Umbra, for example, manages the state for queries in two different regions, thread-local and global state. The former is used for intra-pipeline processing and is thus not relevant to our approach. The global state, on the other hand, holds all data shared between pipelines, such as materialized results, and allows us to access them from the database. We will discuss the access in detail in Section 4.4.

2.2 State Model

After describing the goal and system model for state separation, we can define what *state* is relevant for extraction. As we discussed in Section 2.1.2, the intermediate results of a query materialize its progress. Thus, we only have to focus on these results. This allows us to remove all database-wide information, like indices, from our consideration, as this information is available to all nodes in a cluster in a uniform fashion.

Intermediate results are commonly materialized in blocking operators, i.e., at the end of pipelines. Vectorized systems such as MonetDB [10] materialize results in every operator. In our example in Figure 2, the results are, e.g., materialized in the join hash table after pipeline (1) and the aggregates after pipeline (3). Further, we abstract from the physical representation of the result, such as hash tables. This physical representation can vary greatly between systems and even between different instances of the same system. For example, even switching from a hash join to a blockwise-nested-loop-based join implementation for the same query and system will change the materialization of the results, even though the results will be identical. Therefore, we only consider tuples in the materialized results, not their surrounding index structures. Finally, we only consider results that are still required for query processing. After pipeline (2) has finished, the join hash table produced by pipeline (1) is no longer required and thus not considered part of the state. The results of pipeline (2), in turn, will be used by both pipelines (3) and (6). It is, therefore, part of the query state until both have finished. To summarize, we define query state as follows:

DEFINITION 2. **Query State:** *The query state comprises all tuples materialized within the blocking operators of finished pipelines connecting to not yet finished pipelines.*

We will assume this definition when speaking of *state* in the remainder of this paper. After pipelines (1) to (4) are finished in Figure 2, for example, the query state would contain all tuples in the *id, sum(price)* aggregation, as well as all tuples in the *max ≤ revenue* join hash table.

3 STATE ANALYSIS

Having defined what constitutes the state of a query, we can now analyze the state of typical OLAP workloads. We base this analysis on the well-known TPC-DS OLAP benchmark [52], which models a warehouse for a decision support system. To represent medium-sized workloads, we choose scale factor 100 (SF100), which roughly equals 100GB of data. We analyze the state of each of the 103 TPC-DS queries after every pipeline in query plans generated by the Umbra database system [40]. TPC-DS distributes queries uniformly. Therefore, we include all queries and variants once in our analysis. As the state is comprised of only required tuples and columns in SQL-defined data types, the state size only depends on the query plan and join order, and not on the system used.

3.1 State Size Distribution

As the first analysis, we look at the distribution of state sizes occurring throughout all queries. For this, we measure the size of tuples stored after each pipeline and the number of blocking operators materializing these tuples. Figure 3 displays the results. One can
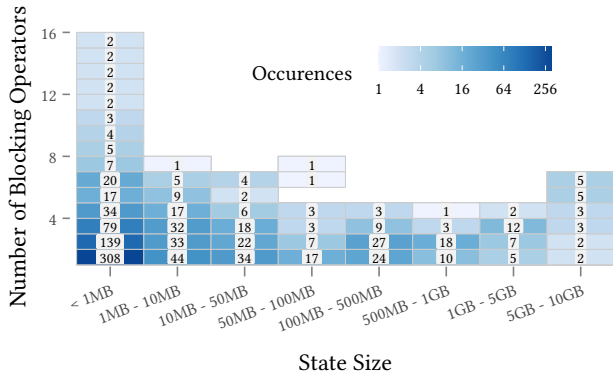
Figure 3: Distribution of state sizes occurring within TPC-DS SF100 by the number of blocking operators involved.
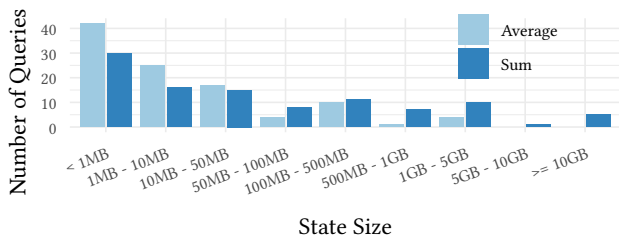

Figure 4: Comparison of the distribution of average and total state sizes per query for TPC-DS SF100.
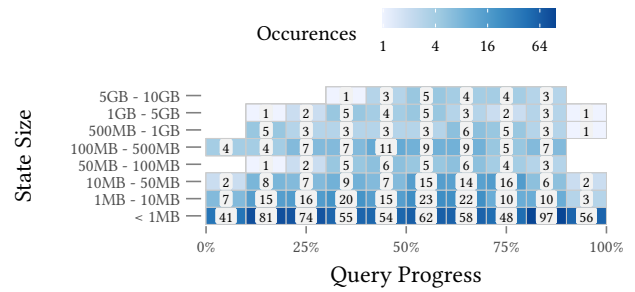

Figure 5: Distribution of state sizes occurring within TPC-DS SF100 by query progress.

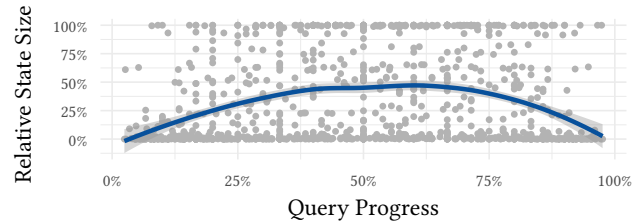**Figure 6: Development of relative state size during queries. State size is relative to the maximum state size reached for a query.**

see that the vast majority of states comprise few operators with less than ten megabytes of data. While most states are small, several states are larger than five gigabytes. Forty-six states exceed one gigabyte in size, while 626 are smaller than 1MB. Overall, 30% of states contain a single blocking operator with less than 1MB of data, and 86% of states do not exceed 100MB. Even though the median state size is only 133KB, the mean state size is 265MB. While 90% of states comprise fewer than five operators, up to 15 operators are involved for some queries.

All intermediate results of a query are relevant for state-separating architectures as the state has to be synchronized after every task. Therefore, we also look at the sum of intermediate result sizes occurring for each query. Figure 4 depicts the distribution of average and total state size per query. One can see that the total state size far outweighs the average. Compared to the mean size of a single state, the mean of all states is 2.6GB, and thus, 10× larger. This sum of state sizes is an upper bound for the state of a query, as it can contain the same pipeline result multiple times. For our example in Figure 2, the result of pipeline ②️ is part of all states starting from pipeline ③️. However, entirely excluding these duplicates would be inaccurate as well, as they have to be transferred to workers multiple times. E.g., the result of pipeline ②️ is required by workers for pipelines ③️ and ⑥️.

3.2 Influence of Query Progress

Given the overall state distribution, we want to analyze further if and how the state distribution relates to query progress. Therefore, we must first define query progress for our model. As we are only interested in the states occurring after pipelines, we define the progress metric based on pipelines only:

DEFINITION 3. **Query progress** $= \frac{\text{\# of finished pipelines}}{\text{\# of total pipelines}}$

While this does not account for the runtime of individual pipelines, it considers the task-based scheduling of cloud jobs. The distribution of state sizes along this query progress for all TPC-DS queries is shown in Figure 5. While large states seem to occur less frequently close to the start and end of queries, the overall distribution shows no significant trends. Both large and small states can occur during every phase of query execution. However, this distribution could be skewed by a few queries with a large state. Figure 6 displays the distribution of state size normalized to the maximum state size for each query to account for this skew. One can see that the trend partly revealed in Figure 5 is more apparent here. On average, the state size grows until around 40% of pipelines are completed and plateaus until around 70%. From there on, the state continuously shrinks. The structure of query plans can explain this. In the beginning, queries collect a lot of data, e.g., in join build sides. In Figure 2, e.g., the first two states include only a single pipeline result. From there on, at least two results are part of the state: at least one join build side and the state of the probe pipeline. After pipeline ⑤️, the state is maximal with three materialized pipelines results (②️, ④️, and ⑤️).

3.3 Discussion

In this section, we have shown the overall distribution and trends in the query state of all TPC-DS queries. However, we have not discussed the implications of state separation arising from this data. Overall, the state sizes in Figure 3 are promising for state separation. Assuming a 10Gbit/s network connection between servers, a round trip for the mean state size takes only 424ms. Still, a complete round trip after every state can add up quickly. In the worst case, up to 9.1GB must be transmitted for a single state, resulting in a 14.6 second round trip. For these large states, transfer time alone can already exceed the execution time of local queries, making re-execution in case of failure more profitable than state separation.

When considering all states occurring for queries (cf. Figure 4), the potential network overhead increases further. Examining the sum of state sizes, the mean of 2.6GB and a maximum of 162.8GB lead to a 4.2 and 261 second round trip, respectively. Nevertheless, given that 86% of single states can be transferred to other workers in less than 160ms, state separation of single states can be profitable for the vast majority of queries.

Evaluating the distribution of state sizes during individual queries in Figure 6 shows that state separation is best early on or close to the end of a query. However, as the relative cost of a restart increases with query runtime, migrating the larger states occurring between 25% and 75% progress might still pay off.

4 ON-DEMAND STATE SEPARATION

The advantages of state separation are well known for cloud environments. Being able to add and remove workers and handle worker crashes offers the flexibility desired by customers. However, synchronizing the state over the network can be expensive, especially for single-worker queries. It is not necessary for those to shuffle state to workers between tasks, and thus, every network transfer is overhead in query execution. As shown in Section 3, sending every state over the network instead of only one can make a 10× difference on average. Furthermore, our approach can optimize for local execution, generating no execution overhead when no state separation is required. While on-demand state separation and migration solutions exist based on virtual machines (VMs), these treat the VM as a blackbox. Therefore, they either have to restart tasks [61] or migrate the entire VMs memory state [45, 48], which is bound to be much larger than just the query state. Approaches that use extensive knowledge about the inner state of queries [44, 60] rely on pre-defined checkpoints. These must be defined before a task starts and cannot be created retroactively on demand.

To achieve a minimal migration state without the need for less flexible checkpoints, we propose scanning and extracting the currently materialized query state, as defined by Definition 2. As this state is part of the execution process, it is accessible at any time without prior preparation. In the remainder of this section, we will describe the high-level design of our approach and the prototypical implementation within the Umbra system using the exemplary use case of query migration:

DEFINITION 4. *Query Migration. Query migration is the process of moving the processing of a query q from an executing server A to a server B without losing the progress achieved for q on A.*
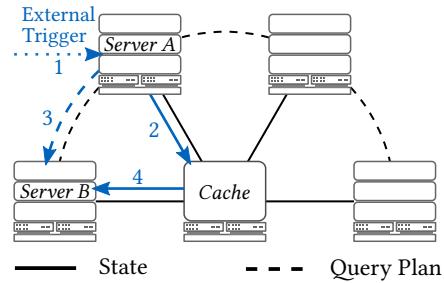

Figure 7: Server cluster for on-demand state separation and query migration. State is only shared via a network cache, query plans can be migrated peer to peer. Exemplary data flow for a migration from server A to server B is highlighted in blue.

4.1 Deployment Environment

The deployment environment is of great importance to enable on-demand state separation, especially in the presence of transient compute resources. We show a possible server configuration in Figure 7. In order to keep the state held in finished pipelines when a worker fails or is taken away, it has to be kept in a durable, external location. Therefore, when state separation is required, all tuples that are part of the state must be cached externally. As state sizes can reach gigabytes (cf. Section 3), all workers (e.g., servers A and B) must access the cache through high-bandwidth connections.

Furthermore, the workers have to communicate to transfer the accompanying query plan. In our current setup, this is realized by peer-to-peer connections between workers. A peer-to-peer setup is the most lightweight option for coordination, requiring few messages and no additional servers. It is, thus, ideal for small and stable deployments. However, it is also possible to handle these transfers using a dedicated coordinator in larger deployments. While this adds communication overhead and requires an additional server, it offers greater flexibility. For example, a coordinator can monitor the running instances to detect migration needs and deal with servers joining and leaving the cluster. While Figure 7 only focuses on those servers and components relevant for state separation and query migration, real deployments will also include additional servers for the cache and storage servers required for storage separation.

There are additional constraints for caches. For one, the workload is different from traditional key-value store workloads. In contrast to those, our data is ephemeral. For query migration, the state is written and read exactly once, often directly after each other. Once the data is read, it is no longer required and, thus, discarded. Furthermore, the state sizes can pose a problem. Many cloud key-value stores limit the maximum value size. The popular key-value database Redis, e.g., has a limit of 512MB [42] for individual values. However, as seen in the state analysis, the state size can reach gigabytes easily. For our workload, the ideal cache would offer high throughput and low latency under high write and read load while offering support for large value sizes. We found the system closest to our requirements to be Apache Crail [47], as it is optimized for ephemeral data and has no limit on value size.

Algorithm 1 Selecting blocking operators contained in a state

```
 1: function SELECTSTATEOPERATORS(finishedPipelines, dependencies)
 2:     dependents ← invert(dependencies)
 3:     stateOperators ← ∅
 4:     for p ∈ finishedPipelines do
 5:         anyUnfinished ← false
 6:         for dep ∈ dependents[p] do
 7:             if dep ∉ finishedPipelines then
 8:                 anyUnfinished ← true
 9:         if anyUnfinished then
10:             append(stateOperators, p.blockingOperator)
11:     return stateOperators
```

4.2 Process Overview

Having defined the deployment environment, we can describe the outline of our on-demand state separation process. We will use the query migration use case of Definition 4, as it is the most involved. Other possible use cases for state separation, such as deferring execution to prioritize other queries or snapshotting, can be realized with the functionality utilized for query migration. For example, consider the migration of the query in Figure 2 from server *A* to server *B* in Figure 7.

First, the need for migration is detected and reported to the server (1). Migration can be triggered by several events, e.g., the indication by the cloud provider that a transient compute resource is being taken away soon or the availability of a faster or cheaper spot instance. Then, the current state of a query according to Definition 2 has to be identified at server *A* and extracted from structures such as hash tables. Server *A* then transfers the extracted state to the external cache (2). On server *A*, the query plan is then adapted to continue from the current state and transferred to the receiving server *B* (3). Server *B* then compiles the received query plan and continues the execution. Whenever a partial result from the state has to be scanned for the first time, it is fetched from the cache and kept locally (4). Snapshots can be realized by periodically sending the current state and the adapted query plan to the cache. Deferring queries is a special case of snapshotting, as the worker does not need to change. Therefore, the state and modified query plan can also be kept locally, thus saving the cost of network transfer. We will use the remainder of this section to describe the steps above in detail.

4.3 State Selection

Once a server has been notified of a desired state separation, the execution of the current query is halted. Then, we identify all operators that are part of the state. For this, we track the current query progress in the form of finished pipelines throughout query execution. Further, to prune all pipeline results no longer required for execution, we calculate all direct dependencies between pipelines.

DEFINITION 5. **Direct pipeline dependency:** *A pipeline A directly depends on a pipeline B if pipeline A directly requires the result of pipeline B for execution.*

In Figure 2, e.g., pipeline ③ depends on pipeline ②, but not on pipeline ①. Given these dependencies, we can now select those finished pipelines still part of the state. The algorithm for this is

shown in Algorithm 1. First, we invert the dependency mapping to get all dependents for a pipeline (line 2). Then, we iterate over all finished pipelines, identifying those that are part of the state (line 4). If all dependents of a pipeline are finished, the pipeline's materialized result is no longer required and, therefore, no longer considered part of the state (lines 6 - 8). Finally, we collect all blocking operators of finished pipelines with at least one dependent for state extraction (line 10).

Directly after pipeline ④ finished executing in our running example, the finished pipelines contain pipelines ①, ②, ③, and ④. Of those, state selection discards ① and ③ as all their dependents (② and ④, respectively) are already finished. We remember the blocking operators for state extraction for the two remaining pipelines, namely, the grouped aggregation *id, sum(price)* and the *max ≤ revenue* join.

4.4 State Extraction

Having found all operators containing state, we have to extract the individual tuples that comprise this state. While a closer mapping to the current state would be to migrate tuples within index structures, such as hash tables, we extract state as defined in Definition 2. State held in operators is optimized for efficient local processing, e.g., by keeping it in pointer-referenced storage and hash tables. This configuration differs between operators and is hard to serialize for network transfer. Furthermore, optimal state structures might differ between servers. Migrating only tuples allows the target system to re-create this per-operator state in a configuration optimized for the local deployment as if it resided in a table. We first want to highlight the high-level process of this tuple-based state extraction before giving a detailed description of the implementation within our system. In general, we have to distinguish between two different kinds of blocking operators.

The first type is operators that only appear as blocking operators within a query, which we call scan-optimized operators. This category comprises unary blocking operators, such as aggregations, sorting, set operations, and potential specialized operators such as K-Means, window functions, or sampling. These operators are the easiest to extract tuples from, as they already offer functionality to scan all tuples. Such operators are always sources of pipelines using their results. Therefore, they produce all tuples when scanned, allowing us to re-use this scan functionality. In many systems, operators can be scanned repeatably to optimize queries, like the *id, sum(price)* aggregation in Figure 2. Functionality for repeated scans further enables state extraction for snapshots without interfering with query execution.

The second type of operator is those with more complex access patterns, such as joins, which appear as a blocking and as a filtering operator. Hash joins, e.g., are optimized for point accesses on the join predicate and often do not offer functionality for full scans. Fortunately, there are only a few operator types in this category. This category only contains different join implementations with non-linear access optimizations in our system. Nevertheless, joins frequently occur in queries and should be considered for query migration. While these operators are not optimized for full scans, their internal structures often still support such scans. Blockwise-nested-loop joins, e.g., materialize their build-side fully without

State Extraction Queries

CacheTransfer — BuildSideScan — $\bowtie_{id = id}$

CacheTransfer — BuildSideScan — $\bowtie_{0.9 \,*\, max\, <=\, revenue}$

CacheTransfer — $\Gamma_{id,\, sum(price)}$

Cached Subtrees

$\bowtie_{0.9 \,*\, max\, <=\, revenue}$ — $\Gamma_{max(revenue)}$ — $\Gamma_{id,\, sum(price)}$ — $\bowtie_{id = customer\_id}$ (Customer, Orders)

$\bowtie_{id = id}$ — Customer

$\Gamma_{id,\, sum(price)}$ — $\bowtie_{id = customer\_id}$ (Customer, Orders)

Modified Plan

Result — $\bowtie_{id = id}$ — Migrated Table — $\bowtie_{0.9 \,*\, max\, <=\, revenue}$ — Migrated Table — Migrated Table
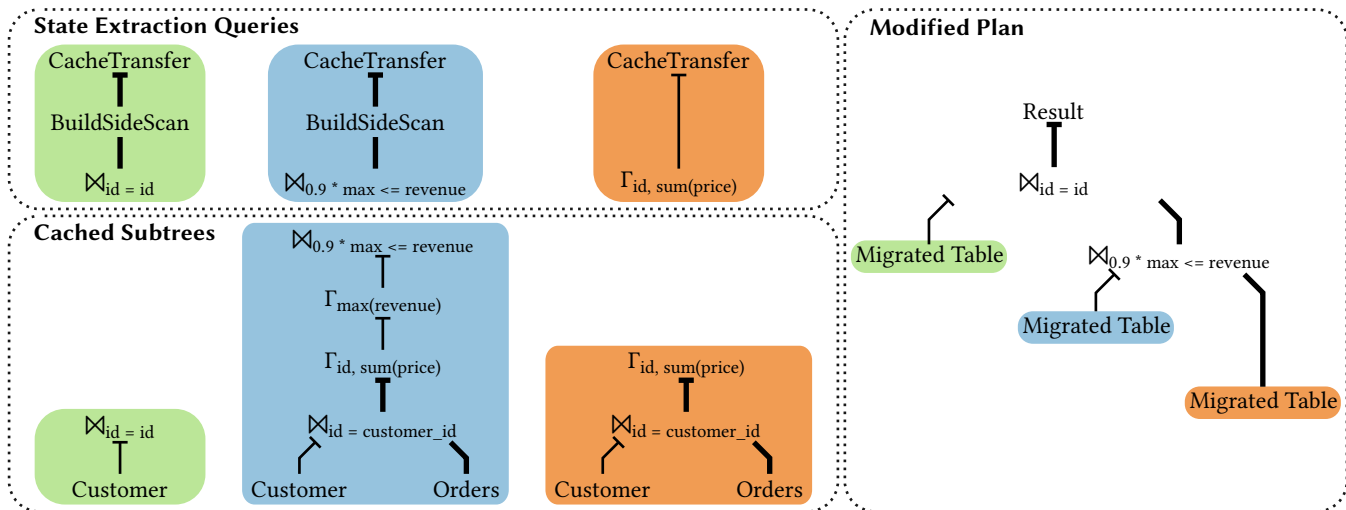
Figure 8: Query migration artifacts of the query in Figure 2 when migrating after pipeline 5. Top left: State extraction queries to be run on the migration source. Bottom left: Extracted state represented by the corresponding subtree, held in the cache. Right: Modified plan to continue execution on the target server.

additional indices, making scans easy. Most index structures, such as hash tables and trees, can be scanned efficiently, allowing us to support the migration of all operators currently implemented within our system.

In the following, we will describe the implementation of query migration for both scan-optimized and index-optimized operators. As the implementation of operators varies heavily between systems, we will limit the implementation to the Umbra [40] system. We outline the requirements and possibility for state extraction in other systems in Section 2.1.2.

4.4.1 Implementation. To best utilize existing infrastructure within the database, we implement the extraction process as a regular query. All tuples are materialized within operators during execution, often nested in a complex operator state. Extracting this state with a query allows us to re-use existing logic and access paths. Furthermore, this allows us to access all optimizations and features of in-database query execution, such as specialized code generation [39] and morsel-driven parallelism [34]. Especially for scan-optimized operators, the state extraction can be realized almost entirely with existing code and logic, allowing easy integration into an existing system. The extraction query plans differ for scan- and index-optimized operators, which we will describe below. Once we have generated this plan, the remaining steps are identical: The query plan is compiled and given access to the state of the query to be migrated or snapshotted. In contrast to regular queries, our state extraction queries do not report the result to the user. Instead, the resulting tuples, i.e., the query state, are collected in a compact format and sent to the cache. Our approach schedules all extraction queries for immediate execution and exclusively to prevent modifications to the state before the extraction is complete.

4.4.2 Scan-Optimized Operators. State from scan-optimized operators can be extracted using only the extracted operator's logic. For this, we duplicate the existing operator into a new query plan

and link the copy to the state of the operator selected for extraction. An example of such an extraction plan for a scan-optimized operator can be seen in Figure 8 for the *id, sum(price)* aggregation. We can again use the fact that scans of an operator's state are non-destructive and reference the state of the existing operator, avoiding a costly copy of the whole operator state.

4.4.3 Index-Optimized Operators. Index-optimized operators require a more in-depth analysis of the state to extract tuples. While it would be possible to generate extraction plans using only existing query logic, e.g., by modifying joins to run against a single tuple that joins with all build-side tuples, we opted to implement dedicated extraction operators instead. Using dedicated operators, we can often bypass the operator's access paths and directly access the data for a scan. This more efficient access strategy comes at the cost of implementing extraction logic for all index-optimized operators. However, as stated above, there are only a few operators in this category that often occur. For space considerations, and because all these extraction operators follow a similar pattern, we will not detail the implementation for every operator. Instead, we will describe the high-level implementation based on a hash join.

Consider, e.g., the *id=id* join extraction in the top left corner of Figure 8. In it, we need to extract all tuples stored in the build-side hash table of the hash join. All operator states are well-defined within Umbra. Therefore, we can locate the hash table from the operator state and make it accessible to our build-side-scan operator. This scan operator then loops over all buckets, extracting all key-value pairs stored within to recreate the tuples. All other specialized extraction operators in our system follow this pattern of accessing the structure holding tuples in the operator's state to be extracted. Again, this scan is non-destructive, and therefore, we do not have to copy the hash table to extract tuples.

Algorithm 2 Modifying the query plan to use the extracted state

```
 1: function MODIFYPLAN(stateOperators)
 2:     opsToExtract ← ∅
 3:     for op ∈ sortPreOrder(stateOperators) do
 4:         if isIndexOptimized(op) then
 5:             toReplace ← op.buildSide
 6:             replaceIn ← op
 7:         else
 8:             toReplace ← op
 9:             replaceIn ← op.parents
10:         migratedTable ← buildTable(toReplace.types)
11:         for location ∈ replaceIn do
12:             if location.isValid() then
13:                 location.replace(toReplace, migratedTable)
14:                 append(opsToExtract, op)
15:     return opsToExtract
```

4.5 Plan Modification

In the final step on the source server, we need to adapt the query plan to incorporate the cached state instead of the subtrees it represents. To achieve this, we modify a copy of the existing query plan. Algorithm 2 displays the pseudo-code for this query plan modification. For every operator in the state, we again have to differentiate whether it is scan- or index-optimized in the current query plan. Index-optimized operators are not the blocking operator of the final pipeline passing through them. Therefore, we cannot replace them entirely with the tuples contained in their state. Instead, we mark the build-side child for migration in the operator itself (line 5). For simplicity, we only consider binary operators with one build side in Algorithm 2. The procedure for n-ary operators is orthogonal, replacing all finished pipelines ending at the state operator with the corresponding state. In Figure 8, this can be seen for the *id = id* and *max ≤ revenue* joins. Our approach replaces only the build side subtrees and not the entire joins in the modified plan.

Scan-optimized operators can only be part of the state operators if all their inputs are finished. Therefore, we can replace the entire operator with the state held within (line 8) without losing progress. However, in contrast to index-optimized operators, it is possible that we need to replace the operator in multiple places as scan-optimized operators can be scanned multiple times within the same query. The *id, sum* aggregation of Figure 2, e.g., is scanned twice. Therefore, a migration after pipeline ② needs to replace it in both parent pipelines ③ and ⑥. One can see that this can lead to conflicting replacements: For example, in the migration displayed in Figure 8, the *id, sum* aggregation is part of the state, and thus, replaced in both parents. However, one parent is further replaced in the *max ≤ revenue* join's build side. For such cases, we always want to ensure only the topmost replacement takes place, as it preserves the most progress. To achieve this, we perform replacements top-down by sorting the state operators (line 3) and always check whether the replacement location is still valid, i.e., contained in the query plan (line 12). This way, replacements will always be optimal, as the topmost operator is considered first. Replacements in the lower part of the tree will either be performed later on or will not occur if the location is no longer valid. To prevent needless network

transfers, we only extract state from operators that are part of the final query plan, i.e., all operators part of a valid replacement (line 14).

4.6 Query Migration and Continuation

Once we extracted all tuples that are part of the state and have generated a query plan utilizing this state, the query can be sent to the desired target. In the scenario outlined above, migrating a query from server *A* to server *B*, neither the state nor the query are initially available at the destination server. In the first step, server *A* sends the modified query plan to server *B* and then aborts the local execution. In turn, the query plan is compiled and executed on server *B*. Whenever the execution reaches the first scan of a migrated table, the table is fetched in parallel from the network cache and held locally for potential subsequent scans. In the case of a migration, the cache can discard each stored value after the first read.

Pausing a query works orthogonally without the need for network transfers. Instead of caching the query plan and state externally, our approach would materialize them in the memory or persistent storage of the worker. Once both are materialized, we abort the query locally to free all working memory for the prioritized query. When the prioritized query finishes, we load the plan from disk and continue its execution. Finally, snapshots register both the state and query plan with the cache. Once all data is cached externally, execution continues on the local server.

4.7 Applications

So far, we have focused on migrating queries between servers. Migration alone already offers several benefits. It can save cost by utilizing cheap spot instances without risk and improve performance by changing to better-suited instances at runtime. However, we understand on-demand state separation as a toolkit that can also be applied in other scenarios. First, as already discussed in the previous subsection, our approach allows users to suspend queries cheaply to prioritize latency-sensitive tasks when compute resources are limited. The snapshotting mechanism of Section 4.6 can be used to deal with worker failures, which we have not discussed so far. In the after-the-fact query migration use case, we rely on prior notice to migrate, which is unavailable in the case of crashes. In order to avoid restarts, this mechanism can be used to create periodic snapshots of a query. In case of a failure, we assign the latest snapshot of the query plan to a new worker, which again fetches migrated tables on demand and continues execution. Furthermore, applications of our approach are not restricted to single-worker queries alone. The extracted state caches independent subtrees of a query, as can be seen in Figure 8. Thus, these subtrees could be executed in parallel on different workers and combined using the steps outlined in this section, effectively enabling scale-out for existing systems.

While they are the focus of our work, possible applications of on-demand state separation are not limited to distributed settings. Materializing tuples with information about their corresponding subtree (cf. Figure 8) can be used to share and re-use intermediate results with other queries [25, 43]. Further, our approach can be used to re-plan queries in the event of network delays [6, 53] or cardinality misestimation in the optimizer [8, 37].

5 EVALUATION

Our evaluation is twofold. In the first part, we provide an in-depth analysis of the amount and sources of the overhead of on-demand state separation on query processing in a series of microbenchmarks. In the second part, we demonstrate the feasibility of our approach for typical cloud use cases. We conduct all experiments in this section using our approach within the Umbra database system [40].

5.1 Setup

To emulate a cloud environment, we run all experiments in this section in a cluster of 4 nodes. Each node is equipped with an Intel Xeon CPU E5-2660 v2 (2.20GHz) and 256GB of DDR3 RAM. The nodes connect to the cluster through a Mellanox ConnectX–3 VPI network interface card (up to 56Gbit/s FDR Infiniband) via a Mellanox SX6005 switch. While an RDMA Infiniband configuration would be most performant, many cloud providers rely on Ethernet connections between servers. For this reason, our implementation uses the TCP network stack as well, and we configure our cluster to run on IP-over-Infiniband (IPoIB) instead of full-fledged Infiniband to emulate a more typical cloud setup.

Two nodes act as source and target servers for query migration, which is the main focus of this evaluation. Each server runs an Umbra instance on a local copy of the TPC-DS SF100 database held in an in-memory file system. This way, we guarantee equal access to the base data, simulating storage separation. The two remaining nodes form the Apache Crail-based network cache [47], with one acting as a namenode and one acting as a datanode. While Crail offers an optimized RDMA-based mode, we again opt for a TCP-based infrastructure to better simulate a typical cloud setup.

5.2 Microbenchmarks

On-demand state separation comprises many individual steps, as outlined in Section 4. Before demonstrating the feasibility through end-to-end benchmarks, we first want to analyze the sources of the introduced overhead in these steps. The two main categories in our analysis are network and execution overhead. The first arises from the topology of the cluster setup and external components, such as network caches, which we cannot directly influence. Overhead stemming from our approach is mainly execution-based, that is, analyzing and extracting the state locally and continuing execution on the remote server.

Configuration. For all experiments in this section, we report overheads based on an average of 5 runs. To provide a detailed analysis, we measure the individual runtime of all sub-steps of migrating after every pipeline occurring in the 103 TPC-DS queries. Further, we perform full migrations and configure both the source and target server to run an identical configuration of Umbra, thus minimizing any configuration influence on runtime. However, we still detected runtime variance in preliminary experiments for local-only and migrating runs, even with identical configurations. Thus, we report overheads as a percentage of the runtime of an entire migration.

Execution Overhead. In the first microbenchmark, we want to highlight the overhead caused by our approach. Multiple factors comprise this overhead: On the source server, this includes state selection (Section 4.3), plan modification (Section 4.5), and compiling state extraction queries, as well as running the extraction up to, but
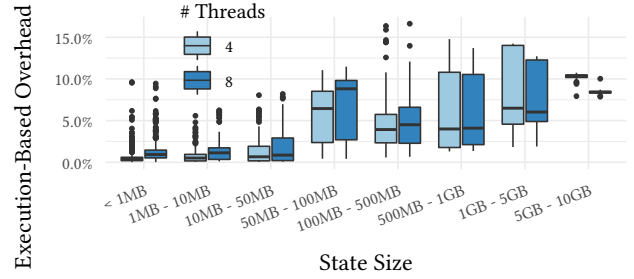

Figure 9: Execution overhead by state size when migrating TPC-DS queries.

excluding, network transfer (Section 4.4). Further, the execution overhead includes parsing and compiling the received query plan on the target server. We compare the overhead generated solely by our approach for two server configurations. Once Umbra is allowed to use up to four worker threads, once up to eight. Figure 9 shows the resulting overheads.

One can see that there is a trend along with the state size for both configurations. When migrating larger states, the overhead grows as well. We expect this increase, as all tuples must be scanned at least once for extraction when materializing them for network transfer. Furthermore, there is no apparent difference between four and eight threads in terms of overhead, indicating that our parallel extraction scales as well as Umbra's query execution framework. This scaling further shows the benefits of utilizing extraction queries in our approach, through which we gain access to parallelism and scheduling optimizations already present in the database.

For both configurations, one can see several outliers for small state sizes. These are primarily from small and fast queries with execution times in milliseconds, where execution does not fully amortize the cost of compiling extraction queries. Nevertheless, on average less than 11% of overall query runtime is spent processing state extraction and migration, independent of server configuration and state size. For states smaller than 50MB, which make up 83% of all states, the mean overhead does not exceed 1.9% independent of the server configuration.

Network Overhead. Having analyzed the processing overhead caused directly by our approach, we want to analyze the overhead caused by the necessary network transfers. While this overhead does not stem from our approach directly, and we thus cannot influence it within our system, it is crucial to understand the overall cost of on-demand state separation. The network overhead measured here is the time required to send and receive the extracted state to and from the cache. Again, we compare the overhead for migrations between two instances with an equal number of worker threads and display the resulting overhead for all migrations in Figure 11.

Overall, the network overhead again clearly grows with the state size migrated. We expect this growth, as network bandwidth is limited and slower than local processing of tuples within a query. However, this overhead is less linear than we have seen for local processing, reaching an average of 45% for states between five and ten gigabytes when using eight worker threads. Furthermore, in
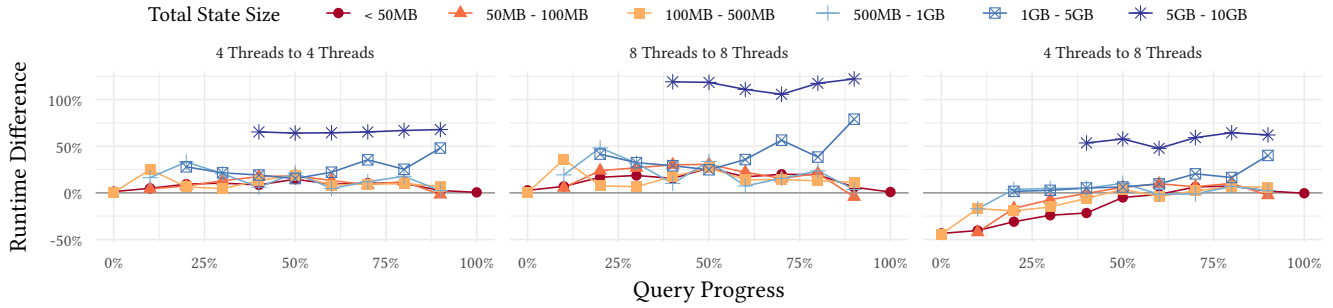
Figure 10: Execution time difference of query migration compared to local execution for 3 server configurations. First thread count denotes source worker threads, execution times without migration are measured on source server.
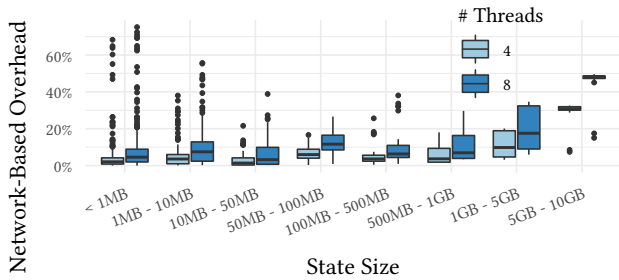

Figure 11: Network overhead by state size when migrating TPC-DS queries.

contrast to the execution-based overhead, one can see that there is a noticeable difference between the configurations. The network transfer is not limited by the compute resources available but by the network bandwidth and latency. Even though network overhead exceeds processing overhead for most state sizes in both configurations, the mean overhead for states smaller than 50MB does not exceed 11% of query runtime.

5.3 Query Migration

Given the individual overheads from the microbenchmarks, we investigate how this translates into the cost of end-to-end query migrations compared to local execution. In addition to migrating between identically configured servers, we further investigate the advantage of fixing an adverse query-to-worker matching by migrating to a more powerful server. Furthermore, we highlight the advantage of our on-demand separation by comparing it to full-fledged state separation, where the state separation of Section 4 is performed after every pipeline.

Configuration. To capture the total cost of migration, we base all experiments in this section on end-to-end query runtime. Query runtime includes every step of query processing, from receiving the SQL query to fully reporting the result, either locally or, in the case of migration, on the remote server. We again migrate all 103 TPC-DS queries and report the average of five runs while re-using the

server configurations from Section 5.2. Unless stated otherwise, all experiments in this subsection report the relative runtime difference between migrating a query and local-only execution on the source server. We show all states < 50MB as a single group for better visibility as they behaved almost identically in all experiments. To better highlight trends, we round query progress to the nearest 10% and report averages within this interval throughout this subsection. **Symmetric Migration**. In the first two experiments, we focus on the overall cost of migration. For this, we migrate between identical instances of Umbra for configurations with four and eight worker threads. This experiment simulates a transient worker being taken away and replaced by another at any part of the query. The results are displayed left and center in Figure 10. We again compare the results for different state sizes. One can see that states smaller than 1GB behave similarly, independent of server configurations. However, there is a significant difference between the two configurations, even for smaller states. We attribute this to the differences in network overhead, which we already identified in Figure 11. In addition to the state size, the migration point also influences the overhead. One can see that migrating between 30% and 80% of query progress is slightly more expensive than at the beginning and end of a query, even when state sizes are similar. We found that states in the middle of a query comprise more operators on average, leading to an increased overhead for compiling and managing state extraction even when the resulting state is of a similar size.

Most query migrations cause less than 25% overhead, making migrating queries more profitable than restarts right from early on. On average, migrating states smaller than 1GB causes 9.3% overhead when using four worker threads and 16.4% when using eight. However, it seems that migrating states larger than 5GB is seldom profitable, especially for the eight-thread configuration. The explosion in overhead for states between one and five gigabytes is caused by only two queries that are no longer compensated for by other states for progress >50%. It is evident that the migration of large states is rarely profitable and will be outperformed by restarts. However, this does not mean that on-demand state separation cannot be profitable for queries with large states. Because state scans are non-destructive, it is possible to extract older, potentially smaller states at the cost of some progress loss. This way, restarts are only required if a query does not have any small states.
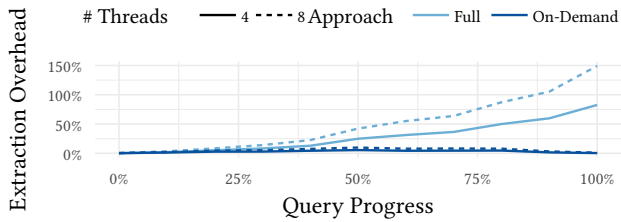
Figure 12: Extraction-caused runtime overhead for full and on-demand state separation for TPC-DS.

Figure 13: Migration latency when executing multiple queries in parallel.

Migrating to Better Instance. On-demand state separation can not only be utilized to migrate between identical instances. It further allows using more powerful instances when they become available, e.g., through spot instances or servers finishing their current query. Utilizing such instances promises the potential to speed up query processing. To investigate the benefits of migrating running queries to faster servers, we migrate from an Umbra instance with four worker threads to one with eight workers. The results, displayed on the right in Figure 10, show that utilizing a faster instance can speed up query processing in many cases, especially for smaller state sizes. As expected, migrating early will lead to the biggest speedup in processing because the faster compute can be used the longest. However, there are instances where migration pays off in every execution phase. Even when 90% of the query is completed, some small states' migrations are still beneficial. Migration is not the only possibility to leverage faster workers. In addition, one needs to consider restarting queries on the new worker.

Ideally, restarting on a worker with twice the compute power will speed up query processing by a factor of two as well. On-demand state separation can outperform such query restarts for many queries in our experiments. On average, migration outperforms restarts once a query reaches 30% progress. When a query has progressed more than 50% on the source server, migration is 67.5% faster when compared to a restart on the destination. Again, larger states are not profitable for on-demand state separation, and query restarts would outperform them throughout the experiment.

On-Demand vs Full State Separation. Having analyzed the cost of on-demand state separation for migrations, we want to compare it to full state separation. Full separation extracts state and synchronizes it with a cache after every pipeline. We compare the average cost of on-demand state separation with the cumulative cost that state separation after every pipeline will have incurred so far. For both approaches, state separation is performed in Umbra as outlined in Section 4. While this leads to a larger state than necessary (cf. Section 3.1), and approaches with full state separation could optimize for smaller states, the overall trends will prevail. The results of each blocking operator still have to be transferred at least once. In contrast to the previous experiments, the runtime overhead no longer includes a full migration, which would disproportionately affect full state separation. Instead, the overhead comprises the work required to extract the state at the migration source only.

Figure 12 shows the resulting overheads in runtime . The execution time overhead shows the advantage of on-demand separation.
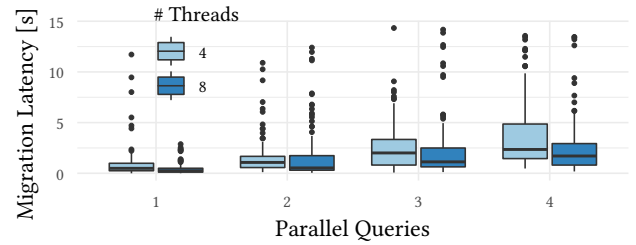
While the overhead grows for full migration with query progress, the overhead of a single migration is almost constant throughout. A full migration causes more than 100% overhead for eight threads, whereas the overhead of on-demand separation never exceeds 10%. The influence of the number of worker threads identified in Figure 10 prevails for both on-demand and full state separation, even when only considering the overhead at the source. The more powerful the servers, the higher the overhead of state separation.

In environments where sudden worker failures are common, the increased cost of full state separation may pay off. However, we argue that the benefits of after-the-fact on-demand separation will outweigh the risk of losing progress in most deployments.

Migration Latency. Finally, we want to demonstrate the capabilities of our approach in real-world applications. We investigate an exemplary use case of vacating a spot instance running multiple queries, e.g., when the cloud provider indicates that they will take it away soon. It is critical to react quickly to a migration request in this scenario. Therefore, we will investigate the migration latency, the time from the notification until the current server is fully vacated, and all progress is held externally. While we here migrate all running queries, it is, of course, also possible to extract only a few queries for load balancing in multi-tenant scenarios [33]. We investigate the latency by running randomly-selected queries in parallel in a loop and triggering migration after a randomly selected duration between 10 and 30 seconds. The migration latency reported is the time from triggering the migration until every query's state and plan are sent to the cache and target server, respectively. Furthermore, to not lose progress, all in-flight tasks, i.e., pipelines, are finished before migrating, which is also included in the latency.

Figure 13 shows the migration latency for 4 and 8 worker threads for 100 runs per thread and query count combination. We removed 19 outliers for better visibility. Of course, one can see that there is linear growth with an increasing number of parallel queries. However, the gap between 4 and 8 threads shrinks for more queries. This shrinkage is again attributable to the network limitation already identified in Figure 11. Furthermore, as we optimize for keeping all progress, the latency includes finishing the current task. It could be further reduced if faster migration is valued over progress kept. On average, even with this additional delay, it takes less than 2.6 seconds to vacate a server, allowing us to react quickly to changes in dynamic environments.

6 RELATED WORK

As more data moves to the cloud, ample research has focused on optimizing data processing for this distributed and flexible environment. This section will provide an overview of the research most essential and relevant to our on-demand state separation approach.

Cloud-Optimized Database Architectures. Cloud-optimized databases, such as Snowflake [12], Google BigQuery [3, 38], and Amazon Redshift [24] optimize for massive parallelism for queries on ever-growing data. Some surveys [49, 67] investigate the challenges and opportunities for databases in cloud environments. Analogous to cloud-optimized databases, we strive to increase flexibility for analytical queries in the cloud. In some systems, this flexibility is enabled through disaggregated storage [11, 12, 16, 41, 54, 56, 66]. Dremel [38], e.g., employs storage disaggregation, as well as memory disaggregation through a shuffle layer, for flexiblity and scalability. We see storage disaggregation as one pillar of flexibility in our approach. To further improve the performance of storage-separated systems, Yang et al. propose a combination of caching and pushing compute to storage to reduce network cost [62]. We also minimize network overhead by migrating a minimal query state. In addition, modern big data systems [31, 51, 65] offer flexibility by directly accessing tables stored in remote storage [4, 19].

Building on the ideas of storage-separated architectures, Aguilar-Saborit et al. [2] describe the state-separating POLARIS system. In addition to storage, they further keep the query state externally, thereby enabling intra-query worker changes. We build upon this idea for our approach. However, we keep state externally only when necessary, thus reducing network overhead. Keeping state in the form of intermediate results externally for transient compute resources has also been proposed for Apache Spark [60, 61]. To materialize intermediate results, Stuedi et al. [30, 47] propose a data store optimized for temporary data in distributed settings.

Adaptive Query Processing. Ample research has been conducted on modifying query execution during runtime in the context of adaptive query processing [7, 14, 23]. While this area focuses on adapting the query plan at runtime, and we currently do not modify execution order when migrating, we still share similar ideas. For example, Xing et al. [59] discuss migrating processing on the fly for load balancing in streaming engines. Orthogonal to our work in case of migration, some works have focused on keeping progress in the case of plan changes in ETL MapReduce [27] and traditional database systems. Keeping progress has been described using artificially introduced operators [8] and unary blocking operators [37] for database systems. These works re-use intermediate results on the same system and do not consider network transfer. To mitigate network dealys, Urhan et al. [6, 53] propose query scrambling.

Instance Migration in Cloud Environments. Many cloud providers offer transient compute resources to customers to increase resource utilization within their datacenters. Often, such transient workers, e.g., spot instances in Amazon AWS, are cheaper than reserved instances. Therefore, utilizing such transient resources has been the focus of recent research. Kraska et al. [32] analyze deployment strategies for fault tolerance mechanisms, such as query restarts and checkpoints. These checkpoints often comprise the entire VM and application state [28, 45, 55, 63, 64]. In contrast, we optimize for a small, system-specific query state that can be extracted at any time. Other systems also optimize for a minimized state [9, 44, 60] but rely on pre-defined checkpoints. Yan et al. [60] propose adaptive fine-grained checkpointing for Apache Spark based on recomputation cost and failure probability. Kaulakiene et al. [28] propose migrating tasks to cheaper or more powerful instances using VM snapshots to optimize the cost and runtime of jobs in a cloud setting. We have identified such migrations as a primary use case for our on-demand state separation approach and optimized specifically for the migration of database workloads. While our work focuses on analytical workloads, migrating between servers is also interesting for transactional workloads [13, 18].

Migration of Intermediate Results. The presented idea to migrate partial query results is inspired by past work in multi-engine environments. These so-called polystores span a combination of stream processing, big data, and database systems [15, 20, 22, 26, 36], some surveyed by Tan et al. [50]. While we only consider migrating to other instances of the same engine and optimize for flexibility in our work, works on multi-engine environments focus on a range of optimization criteria. For example, Agrawal et al. [1] optimize performance by selecting a combination of execution engines for a single task and migrating intermediate results between these engines. Simitsis et al. [46] describe an optimizer for data workflows comprising multiple engines, taking both the execution and data shipping cost into account. Focussing specifically on data migration, Dziedzic et al. [17] discuss challenges and solutions for sharing results between engines. While we discuss our approach in the context of flexibility, it can also be used to optimize for different metrics.

7 CONCLUSION

In this paper, we present a novel on-demand state separation approach for data processing in the cloud. In contrast to existing state-separating architectures, our approach can establish state separation after-the-fact, e.g., when migrating between workers. This way, our approach only incurs the overhead of syncing state externally when necessary. To motivate our approach, we provided an extensive analysis of query state occurring within the TPC-DS benchmark, showing that on-demand extraction can reduce the transferred state by an order of magnitude. Our approach exploits existing access paths to extract state with specialized extraction queries, allowing state extraction with minimal code and runtime overhead while utilizing all features of modern query engines.

We demonstrate the feasibility of our approach using an implementation of on-demand state separation in the Umbra database system. The experimental analysis shows that our approach can outperform full state separation and query restarts in many scenarios. Our in-depth cost analysis demonstrates that the majority of overhead stems from network overhead. With the roll-out of more powerful network infrastructure in the future, we expect our approach to be beneficial in even more use cases.

REFERENCES

[1] Divy Agrawal, Sanjay Chawla, Bertty Contreras-Rojas, Ahmed K. Elmagarmid, Yasser Idris, Zoi Kaoudi, Sebastian Kruse, Ji Lucas, Essam Mansour, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, Saravanan Thirumuruganathan, and Anis Troudi. 2018. RHEEM: Enabling Cross-Platform Data Processing - May The Big Data Be With You! -. *Proc. VLDB Endow.* 11, 11 (2018), 1414–1427.

[2] Josep Aguilar-Saborit and Raghu Ramakrishnan. 2020. POLARIS: The Distributed SQL Engine in Azure Synapse. *Proc. VLDB Endow.* 13, 12 (2020), 3204–3216.

[3] H Ahmadi. 2016. In-memory Query Execution in Google BigQuery. *Google Cloud Blog* (2016).

[4] Amazon. 2022. Cloud Object Storage - Amazon S3. Retrieved February 22, 2022 from https://aws.amazon.com/s3/

[5] Pradeep Ambati, Noman Bashir, David E. Irwin, and Prashant J. Shenoy. 2021. Good Things Come to Those Who Wait: Optimizing Job Waiting in the Cloud. In *SoCC*. ACM, 229–242.

[6] Laurent Amsaleg, Michael J. Franklin, Anthony Tomasic, and Tolga Urhan. 1996. Scrambling Query Plans to Cope With Unexpected Delays. In *PDIS*. IEEE Computer Society, 208–219.

[7] Shivnath Babu and Pedro Bizarro. 2005. Adaptive Query Processing in the Looking Glass. In *CIDR*. www.cidrdb.org, 238–249.

[8] Shivnath Babu, Pedro Bizarro, and David J. DeWitt. 2005. Proactive Reoptimization. In *SIGMOD Conference*. ACM, 107–118.

[9] Carsten Binnig, Abdallah Salama, Erfan Zamanian, Muhammad El-Hindi, Sebastian Feil, and Tobias Ziegler. 2015. Spotgres - parallel data analytics on Spot Instances. In *ICDE Workshops*. IEEE Computer Society, 14–21.

[10] Peter A. Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. 2006. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *SIGMOD Conference*. ACM, 479–490.

[11] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *SIGMOD Conference*. ACM, 2477–2489.

[12] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD Conference*. ACM, 215–226.

[13] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2011. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud using Live Data Migration. *Proc. VLDB Endow.* 4, 8 (2011), 494–505.

[14] Amol Deshpande, Zachary G. Ives, and Vijayshankar Raman. 2007. Adaptive Query Processing. *Found. Trends Databases* 1, 1 (2007), 1–140.

[15] Katerina Doka, Nikolaos Papailiou, Victor Giannakouris, Dimitrios Tsoumakos, and Nectarios Koziris. 2016. Mix 'n' match multi-engine analytics. In *IEEE BigData*. IEEE Computer Society, 194–203.

[16] Dominik Durner, Badrish Chandramouli, and Yinan Li. 2021. Crystal: A Unified Cache Storage System for Analytical Databases. *Proc. VLDB Endow.* 14, 11 (2021), 2432–2444.

[17] Adam Dziedzic, Aaron J. Elmore, and Michael Stonebraker. 2016. Data transformation and migration in polystores. In *HPEC*. IEEE, 1–6.

[18] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD Conference*. ACM, 301–312.

[19] Apache Software Foundation. 2021. Apache Hadoop. Retrieved February 22, 2022 from https://hadoop.apache.org/

[20] Vijay Gadepally, Peinan Chen, Jennie Duggan, Aaron J. Elmore, Brandon Haynes, Jeremy Kepner, Samuel Madden, Tim Mattson, and Michael Stonebraker. 2016. The BigDAWG polystore system and architecture. In *HPEC*. IEEE, 1–6.

[21] Panagiotis Garefalakis, Konstantinos Karanasos, and Peter R. Pietzuch. 2019. Neptune: Scheduling Suspendable Tasks for Unified Stream/Batch Applications. In *SoCC*. ACM, 233–245.

[22] Victor Giannakouris, Nikolaos Papailiou, Dimitrios Tsoumakos, and Nectarios Koziris. 2016. MuSQLE: Distributed SQL query execution over multiple engine environments. In *IEEE BigData*. IEEE Computer Society, 452–461.

[23] Anastasios Gounaris, Efthymia Tsamoura, and Yannis Manolopoulos. 2013. Adaptive Query Processing in Distributed Settings. In *Advanced Query Processing (1)*. Intelligent Systems Reference Library, Vol. 36. Springer, 211–236.

[24] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *SIGMOD Conference*. ACM, 1917–1923.

[25] Milena Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo Goncalves. 2010. An architecture for recycling intermediates in a column-store. *ACM Trans. Database Syst.* 35, 4 (2010), 24:1–24:43.

[26] Abdulrahman Kaitoua, Tilmann Rabl, Asterios Katsifodimos, and Volker Markl. 2019. Muses: Distributed Data Migration System for Polystores. In *ICDE*. IEEE, 1602–1605.

[27] Konstantinos Karanasos, Andrey Balmin, Marcel Kutsch, Fatma Ozcan, Vuk Ercegovac, Chunyang Xia, and Jesse Jackson. 2014. Dynamically optimizing queries over large scale data platforms. In *SIGMOD Conference*. ACM, 943–954.

[28] Dalia Kaulakiene, Christian Thomsen, Torben Bach Pedersen, Ugur Çetintemel, and Tim Kraska. 2015. SpotADAPT: Spot-Aware (re-)Deployment of Analytical Processing Tasks on Amazon EC2. In *DOLAP*. ACM, 59–68.

[29] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. *VLDB J.* 30, 5 (2021), 883–905.

[30] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *OSDI*. USENIX Association, 427–444.

[31] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR*. www.cidrdb.org.

[32] Tim Kraska, Elkhan Dadashov, and Carsten Binnig. 2017. Spotlytics: How to Use Cloud Market Places for Analytics?. In *BTW (LNI)*, Vol. P-265. GI, 361–380.

[33] Willis Lang, Srinath Shankar, Jignesh M. Patel, and Ajay Kalhan. 2014. Towards Multi-Tenant Performance SLOs. *IEEE Trans. Knowl. Data Eng.* 26, 6 (2014), 1447–1463.

[34] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD Conference*. ACM, 743–754.

[35] Viktor Leis and Maximilian Kuschewski. 2021. Towards Cost-Optimal Query Processing in the Cloud. *Proc. VLDB Endow.* 14, 9 (2021), 1606–1612.

[36] Harold Lim, Yuzhang Han, and Shivnath Babu. 2013. How to Fit when No One Size Fits. In *CIDR*. www.cidrdb.org.

[37] Volker Markl, Vijayshankar Raman, David E. Simmen, Guy M. Lohman, and Hamid Pirahesh. 2004. Robust Query Processing through Progressive Optimization. In *SIGMOD Conference*. ACM, 659–670.

[38] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proc. VLDB Endow.* 13, 12 (2020), 3461–3472.

[39] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.

[40] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. www.cidrdb.org.

[41] Presto. 2022. Distributed SQL Query Engine for Big Data. Retrieved February 22, 2022 from https://prestodb.io/

[42] Redis. 2022. Redis - Data types. Retrieved February 22, 2022 from https://redis.io/topics/data-types

[43] Timos K. Sellis. 1988. Multiple-Query Optimization. *ACM Trans. Database Syst.* 13, 1 (1988), 23–52.

[44] Prateek Sharma, Tian Guo, Xin He, David E. Irwin, and Prashant J. Shenoy. 2016. Flint: batch-interactive data-intensive processing on transient servers. In *EuroSys*. ACM, 6:1–6:15.

[45] Supreeth Shastri and David E. Irwin. 2017. HotSpot: automated server hopping in cloud spot markets. In *SoCC*. ACM, 493–505.

[46] Alkis Simitsis, Kevin Wilkinson, Malú Castellanos, and Umeshwar Dayal. 2012. Optimizing analytic data flows for multiple execution engines. In *SIGMOD Conference*. ACM, 829–840.

[47] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. 2017. Crail: A High-Performance I/O Architecture for Distributed Data Processing. *IEEE Data Eng. Bull.* 40, 1 (2017), 38–49.

[48] Supreeth Subramanya, Tian Guo, Prateek Sharma, David E. Irwin, and Prashant J. Shenoy. 2015. SpotOn: a batch computing service for the spot market. In *SoCC*. ACM, 329–341.

[49] Junjay Tan, Thanaa M. Ghanem, Matthew Perron, Xiangyao Yu, Michael Stonebraker, David J. DeWitt, Marco Serafini, Ashraf Aboulnaga, and Tim Kraska. 2019. Choosing A Cloud DBMS: Architectures and Tradeoffs. *Proc. VLDB Endow.* 12, 12 (2019), 2170–2182.

[50] Ran Tan, Rada Chirkova, Vijay Gadepally, and Timothy G. Mattson. 2017. Enabling query processing across heterogeneous data models: A survey. In *IEEE BigData*. IEEE Computer Society, 3211–3220.

[51] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. 2010. Hive - a petabyte scale data warehouse using Hadoop. In *ICDE*. IEEE Computer Society, 996–1005.

[52] Transaction Processing Performance Council (TPC). 2021. TPC benchmark DS: Standard specification. Retrieved February 15, 2022 from http://www.tpc.org/

[53] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. 1998. Cost Based Query Scrambling for Initial Delays. In *SIGMOD Conference*. ACM Press, 130–141.

[54] Ben Vandiver, Shreya Prasad, Pratibha Rana, Eden Zik, Amin Saeidi, Pratyush Parimal, Styliani Pantela, and Jaimin Dave. 2018. Eon Mode: Bringing the Vertica Columnar Database to the Cloud. In *SIGMOD Conference*. ACM, 797–809.

[55] William Voorsluys and Rajkumar Buyya. 2012. Reliable Provisioning of Spot Instances for Compute-intensive Applications. In *AINA*. IEEE Computer Society, 542–549.

[56] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *NSDI*. USENIX Association, 449–462.

[57] Benjamin Wagner, André Kohn, and Thomas Neumann. 2021. Self-Tuning Query Scheduling for Analytical Workloads. In *SIGMOD Conference*. ACM, 1879–1891.

[58] Christian Winter, Tobias Schmidt, Thomas Neumann, and Alfons Kemper. 2020. Meet Me Halfway: Split Maintenance of Continuous Views. *Proc. VLDB Endow.* 13, 11 (2020), 2620–2633.

[59] Ying Xing, Stanley B. Zdonik, and Jeong-Hyon Hwang. 2005. Dynamic Load Distribution in the Borealis Stream Processor. In *ICDE*. IEEE Computer Society, 791–802.

[60] Ying Yan, Yanjie Gao, Yang Chen, Zhongxin Guo, Bole Chen, and Thomas Moscibroda. 2016. TR-Spark: Transient Computing for Big Data Analytics. In *SoCC*. ACM, 484–496.

[61] Youngseok Yang, Geon-Woo Kim, Won Wook Song, Yunseong Lee, Andrew Chung, Zhengping Qian, Brian Cho, and Byung-Gon Chun. 2017. Pado: A Data Processing Engine for Harnessing Transient Resources in Datacenters. In *EuroSys*.

ACM, 575–588.

[62] Yifei Yang, Matt Youill, Matthew E. Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2021. FlexPushdownDB: Hybrid Pushdown and Caching in a Cloud DBMS. *Proc. VLDB Endow.* 14, 11 (2021), 2101–2113.

[63] Sangho Yi, Artur Andrzejak, and Derrick Kondo. 2012. Monetary Cost-Aware Checkpointing and Migration on Amazon Cloud Spot Instances. *IEEE Trans. Serv. Comput.* 5, 4 (2012), 512–524.

[64] Sangho Yi, Derrick Kondo, and Artur Andrzejak. 2010. Reducing Costs of Spot Instances via Checkpointing in the Amazon Elastic Compute Cloud. In *IEEE CLOUD*. IEEE Computer Society, 236–243.

[65] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[66] Qizhen Zhang, Philip A Bernstein, Daniel S Berger, Badrish Chandramouli, Vincent Liu, and Boon Thau Loo. 2022. Compucache: Remote computable caching using spot vms. In *CIDR*. www.cidrdb.org.

[67] Qizhen Zhang, Yifan Cai, Sebastian Angel, Vincent Liu, Ang Chen, and Boon Thau Loo. 2020. Rethinking Data Management Systems for Disaggregated Data Centers. In *CIDR*. www.cidrdb.org.