UNIVERSITY OF CALIFORNIA
RIVERSIDE

Query Optimization in the Era of Big Data Management Systems

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Christina Pavlopoulou

December 2020

Dissertation Committee:

Dr. Vassilis J. Tsotras, Co-Chairperson
Dr. Michael J. Carey, Co-Chairperson
Dr. Vagelis Papalexakis
Dr. Ahmed Eldawy
Dr. Vagelis Hristidis

The Dissertation of Christina Pavlopoulou is approved:

Committee Co-Chairperson

Committee Co-Chairperson

University of California, Riverside

Acknowledgments

I would like to express my sincere gratitude to my advisors Dr. Vassilis J. Tsotras and Dr. Michael Carey for their support, patience, and encouragement to explore new research ideas. Besides my advisors, I would like also to thank the rest of my thesis committee: Dr. Vagelis Papalexakis, Dr. Ahmed Eldawy and Dr. Vagelis Hristidis.

My sincere thanks and gratitude to my family who provided invaluable support, encouragement and advise during the whole PhD journey. I would like to thank my parents, Kostas Pavlopoulos and Vana Florou for their unwavering belief in me. Thank you for your unconditional love and understanding. Above all, I would like to thank my husband Dr. Vasileios Zois for his constant love and support, for keeping me motivated all these past few years. Thank you for giving me a place to stand on and move the world.

The text of this dissertation, in part or in full, is a reprint of the material as it appears in Proceedings of the 21st International Conference on Extending Database Technology (EDBT18) (A Parallel and Scalable Processor for JSON Data, Vienna, Austria, March 26-29, 2018) and in the following arxiv preprint: (https://arxiv.org/abs/2010.00728) (Revisiting Runtime Dynamic Optimization in Big Data Management Systems). The co-author (Dr. Vassilis Tsotras) directed and supervised the research which forms the basis for this dissertation. The co-authors (Michael J. Carey, Till Westmann and Preston E. Carman) reviewed and co-wrote in part the aforementioned paper.

To my loving husband Vasilis for his eternal devotion, support, and encouragement.

To my parents, Vana and Kostas, who inspired me to pursue and achieve more.

ABSTRACT OF THE DISSERTATION

Query Optimization in the Era of Big Data Management Systems

by

Christina Pavlopoulou

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2020
Dr. Vassilis J. Tsotras, Co-Chairperson
Dr. Michael J. Carey, Co-Chairperson

Over the past decade, a number of data intensive scalable systems have been developed to process extremely large collections of data. To process such data as efficiently as possible the need for big data query optimization has emerged. This thesis concentrates on two research problems within query optimization for big data.

The first part of this work aims at the use of cost-based decisions to efficiently process massive collections of data. Traditional query optimizers are cost-based and "upfront", using statistical estimates of intermediate result cardinalities to assign costs and pick the best query plan prior to its execution. However, such estimates tend to become less accurate in the presence of filtering conditions due either to undetected correlations between multiple predicates local to a single dataset, to predicates with query parameters, or to predicates involving user-defined functions (UDFs). Consequently, traditional query optimizers tend to ignore or miscalculate those settings, thus leading to suboptimal execution plans. Given the volume of today's data, a suboptimal plan can quickly become very inefficient. To address this, we instead revisit the old idea of runtime dynamic optimization and adapt it to a

shared-nothing distributed database system, AsterixDB. The optimization process starts by first executing all predicates local to a single dataset and continues in stages (re-optimization points). The intermediate result created from each stage is used to re-optimize the remaining query. This re-optimization approach avoids inaccurate intermediate result cardinality estimations, thus leading to much better execution plans. While it introduces overhead for materializing these intermediate results, our experiments with industry benchmark data show that this overhead is relatively small and that it is an acceptable price to pay given the optimization benefits. In fact, our experimental evaluation shows that runtime dynamic optimization leads to much better execution plans as compared to the current default AsterixDB plans as well as to plans produced by static cost-based optimization (i.e. based on the initial dataset statistics) and other state-of-the-art approaches.

The second part of this thesis focuses on heuristic decisions that can improve the querying of JSON data. With the abundance of web data, JSON has become the de-facto data exchange format today. However, querying JSON data is not always efficient, especially in the case of large data repositories. In this part of our work we integrate the JSONiq extension of the XQuery language specification into an existing query processor (namely, Apache VXQuery) so as to enable the querying of JSON data in parallel. In particular, we have implemented three categories of rewrite rules that can efficiently handle path expressions in parallel along with introducing intra-query parallelism. An evaluation of our approach using a large (803GB) real dataset of sensor readings shows that the proposed rewrite rules lead to efficient and scalable parallel processing of JSON data.

Contents

List of Figures

List of Tables

Chapter 1

Introduction & Motivation

In the last decade, a number of Big Data Management Systems (BDMS) have been developed for processing massive collections of data. At the same time, high-level declarative query languages, like SQL and its extension SQL++, gained popularity due to their ease of use and being a more concise way to express queries over data, so BDMS have started to support them. However, a key challenge in this environment is query optimization, which mainly decides the evaluation order of query operators along with their physical implementation. Most big data query optimizers (Jaql [30], Stratosphere [29], DryadLINQ [41], SCOPE [80], Spark [77] and Shark [75]) address query optimization using a static cost-based approach. This includes collecting statistics on base datasets and formulating a complete query plan based on them (static cost-based optimization). Nevertheless, for complex queries (i.e. queries with multi-way joins) statistical information on base datasets is not enough. In particular, the join result size is very difficult to accurately predict (especially in the case of non primary/foreign key relationships), and when the query has multiple

joins, the error will propagate to the rest of the join estimations. The situation can become even worse when base datasets are accessed with multiple predicates with undetected correlations, external variables, and/or user-defined functions (UDFs). In the aforementioned cases, traditional optimizers use default values as described in [68]. However, these values can lead to estimates that are too far away from the actual sizes. All these issues can lead to very inefficient execution plans that, with the increase in the amount of data, can become even worse.

In an effort to address these issues, we revisit a runtime Dynamic Optimization approach introduced by INGRES [74] and adapt it to a BDMS, namely AsterixDB. The main focus of early dynamic optimization is that the execution of the original query can be performed in stages while acquiring more accurate cardinalities for intermediate results. However, this is not enough to build accurate cost models, and thereby more complex statistical information is needed. Hence we enhance the INGRES approach with better statistics. In more detail, we decompose the initial query submitted by the user, breaking it into smaller subqueries (i.e one-variable queries with one dataset in the FROM clause and single-join queries) and we execute them separately. At the same time, we gather detail statistics for the intermediate results. Based on them, we then build cost models that refine the remaining choices of join ordering and algorithms. This re-optimization leads to more accurate intermediate result statistics and thus to better execution plans. Although this approach also introduces an overhead of intermediate result materialization, our experimental evaluation shows that this overhead is tolerable considering the benefits brought by it in forming superior execution plans.

Another important factor in the era of big data is the need for data platforms to communicate with each other in an efficient way. To that end, several data exchange formats have been developed, among which JSON is currently the most popular due to its simplicity. The large volume of data created and exchanged today requires scalable systems to process it. In the literature there have been multiple works on systems trying to efficiently process large amounts of JSON data. However, most of them either need a pre-processing step to support JSON data or a loading stage to store it in their own internal data model formats. Even for systems that can process JSON data directly, they tend to be optimized towards the processing of relatively simple and flat document structures. Nevertheless, for more complex JSON document structures, more sophisticated ways of processing are required.

In this thesis, we aim to process raw JSON data independently of its structure by proposing a set of efficient rewrite rules whose main goal is to eliminate the need for unnecessary buffer space and thereby buffering only the objects that are really needed. Firstly, we introduce a heuristic rule that decreases the buffer size needed between query operators, so large sequences of objects are avoided. The second group of rules focus on further decreasing the initial buffering requirement by not storing the whole JSON document anymore, instead buffering only the qualified objects after the application of query expressions. The last category of rules concentrates on queries that combine the group-by operator and some aggregation function exploiting the time spent on forming the groups to also perform the aggregation. With the incorporation of these rules, we are able to treat complex and nested JSON structures as if they are completely flat and the queried information is directly acces-

sible. Our evaluation shows the superiority of our techniques against other big data systems for JSON.

The rest of this thesis is organized as follows; In Chapter 2, we provide an itroduction of both the AsterixDB and VXQuery frameworks, including an overview of the statistics collection framework in AsterixDB and the core features of the JSONiq language. Chapter 3 describes an INGRES-like runtime dynamic optimization approach along with its integration in AsterixDB. In addition, it discusses how we handle queries involving external parameters, multiple and correlated predicates, and UDFs. Chapter 4 discusses the challenges of querying large amounts of JSON data and proposes some heuristic rules that dramatically improves query performance as proven in the experimental section. Finally, Chapter 5 summarizes the lessons learned from developing the various big data query optimization techniques and provides insights on potential future work.

Chapter 2

Background

This chapter first describes the AsterixDB framework which we used to implement our Adaptive Query Processing methods discussed in Chapter 3. Figure 2.1 shows the system's layered architecture, including the layers of the Algebricks and Hyracks infrastructure. Chapter 4 will present details of how these layers are used to implement several rewrite rules for efficient processing of JSON data using the the Apache VXQuery processor [34]. In Section 2.2 we describe the architecture of the statistics collection framework that is used in Chapter 3 to construct efficient cost models. Finally, we provide a brief explanation of some key features of the JSONiq language which we later use as an insight to design our rules (Chapter 4).

2.1 AsterixDB Background

Apache AsterixDB is a parallel, shared-nothing platform that provides the ability to ingest, store, index, query, and analyze mass quantities of semistructured data. As shown

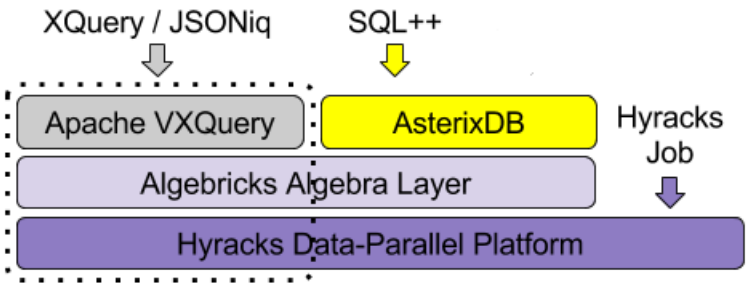


Figure 2.1: The VXQuery and AsterixDB Architecture

in Figure 2.1, to process a submitted query, AsterixDB compiles it into an Algebricks [31] program also known as the logical plan. This plan is then optimized via rewrite rules that reorder the Algebricks operators and introduce partitioned parallelism for scalable execution. After this (rule-based) optimization step, a code generation step translates the resulting physical query plan into a corresponding Hyracks Job [32] that will use the Hyracks engine to compute the requested query results. Finally, the runtime plan is distributed across the system and executed locally on every slave of the cluster. Following is a more detailed description of Algebricks and Hyracks frameworks.

2.1.1 Hyracks

Hyracks [32] is a data-parallel execution platform that builds upon mature parallel database techniques and modern big data trends. This generic platform offers a framework to run dataflows in parallel on a shared-nothing cluster. The system was designed to be independent of any particular data model. Hyracks processes data in partitions of contiguous bytes, moving data in fixed-sized frames that contain physical records. It also defines interfaces that allow users of the platform to specify the data-type details for comparing, hashing, serializing and de-serializing data.

A Hyracks job is defined by a dataflow directed acyclic graph (DAG) with operators (nodes) and connectors (edges). During execution, the operators allow the computation to consume an input partition and produce an output partition while the connectors redistribute data among partitions. The dataflow among Hyracks operators is push-based: each source (producer) operator pushes the output frames to a target (consumer) operator. The extensible runtime platform provides a number of operators and connectors for use in forming Hyracks jobs. While each operator's operation is defined by Hyracks, the operator relies on data-model specific functionality provided by the processor operating on top of Hyracks.

2.1.2 Algebricks

Algebricks [31] is a parallel framework providing an abstract algebra for parallel query translation and optimization. This language-agnostic toolbox complements the lower-level extensible Hyracks platform. Implementations of data-intensive programming languages can extend Hyracks' model-agnostic algebraic layer to create parallel query processors on top of the Hyracks platform. A language developer is free to define the language and data model when using the Hyracks platform and the Algebricks toolkit. Algebricks features a rule-based optimizer and data-model-neutral operators that allow for language specific customization.

A system that uses Algebricks for its query processing provides its own parser and translator to translate a query to a query plan that uses Algebricks' logical operators as an intermediate representation. The Algebricks rule-based optimizer then transforms the query plan over three stages. The first is a Logical-to-Logical plan optimizer that creates alternate logical plans. Once the logical plan is finalized, the Logical-to-Physical plan optimizer

converts the logical operators into a physical plan. Then, the physical optimizer considers the operator characteristics, partition properties, and data locality to choose the optimal physical implementation for the plan. Algebricks provides generic language-independent rewrite rules for each stage and allows for the addition of other rules. Finally, a Hyracks job is generated and submitted for execution on a Hyracks cluster.

Algebricks' intermediate logical algebra uses logical operators that map onto Hyracks' physical operators. A logical operator's properties are considered when determining the best physical operator. In chapter 3, we focus mostly on the physical implementation of the JOIN operator; thus, here we present three different algorithms (Hash Join, Broadcast Join and Indexed Nested Loop Join) supported by AsterixDB currently.

Hash Join: Assuming the join's input data is not partitioned in a useful way, the algorithm redistributes the data by hashing both inputs on the join key(s) – thereby ensuring that objects that should be joined will be routed to the same partition for processing – and then effects the join using dynamic hash join. In more detail, the "build" side of the join is first re-partitioned and fed over the network into the build step of a local hash join; each partition will then have some portion (perhaps all) of the to-be-joined build input data in memory, with the rest (if any) in overflow partitions on disk. The "probe" side of the join is then re-partitioned similarly, thus creating a pipelined parallel orchestration of a dynamic hash join. In the event that one of the inputs is already partitioned on the join key(s), e.g., because the join is a key/foreign key join, re-partitioning is skipped (unnecessary) for that input and communication is saved.

Broadcast Join: This strategy employs a local dynamic hash join where one of the join inputs (ideally a small one) is broadcast – replicated, that is – to all partitions of the other input. The broadcast input is used as the build input to the join, and once the build phase is done the participating partitions can each probe their local portion of the other larger input in order to effect the join.

Indexed Nested Loop Join: Here, one of the inputs is broadcast (replicated) to all of the partitions of the other input, which for this strategy must be a base dataset with an index on the join key(s); as broadcast objects arrive at each partition they are used to immediately probe the index of the other (called "inner") dataset.

Currently, in AsterixDB, the hash join is picked by default unless there are query hints that make the optimizer pick one of the other two algorithms. However, when a broadcast join can be applied, joins can complete much faster as expensive shuffling of the large dataset is avoided.

Furthermore, the Algebricks logical operators exchange data in the form of logical tuples, each of which is a set of fields. The following Algebricks logical operators are used in a basic AsterixDB query plan:

- **EMPTY-TUPLE-SOURCE:** outputs an empty tuple used by other operators to initiate result production.
- **DATASCAN:** takes as input a tuple and a data source and extends the input tuple to produce tuples for each item in the source.
- **ASSIGN:** executes a scalar expression on a tuple and adds the result as a new field in the tuple.

- **AGGREGATE:** executes an aggregate expression to create a result tuple from a stream of input tuples. The result is held until all tuples are processed and then returned in a single tuple.
- **UNNEST:** executes an unnesting expression for each tuple to create a stream of output tuples per input.
- **SUBPLAN:** executes a nested plan for each tuple input. This plan consists of an AGGREGATE and an UNNEST operator.
- **GROUP-BY:** executes an aggregate expression to produce a tuple for each set of items having the same grouping key.

The final layer, either AsterixDB or VXQuery, parses the submitted query into an abstract syntax tree (AST) and is then analyzed, normalized, and translated into a logical plan which becomes the input to the Algebricks framework. For AsterixDB, the final layer supports the SQL++ language, while for VXQuery supports an XQuery processor engine. To build a JSONiq processor and integrate it into VXQuery (the main focus of Chapter 4), we used the JSONiq extension to XQuery specifications. Specifically, we focused mostly on implementing all the necessary modules to successfully parse and evaluate JSONiq queries. Additionally, several modules were implemented to enable JSON file parsing and support an internal in-memory representation of the corresponding JSON items.

2.2 Statistics Collection

To form an efficient execution plan regarding join ordering and algorithm which is the main focus of Chapter 3, we need statistical information on the base and intermediate datasets of a query. These statistics are later used to estimate the actual join result size by using the following formula, as described in [68]:

$$A \bowtie_k B = S(A) * S(B)/max(U(A.k), U(B.k)) \tag{2.1}$$

where $S(x)$ is the size of dataset x and $U(x.k)$ is the number of unique elements for attribute k of dataset x. The size of a dataset is the number of qualified records in the dataset immediately before the join operation. If a dataset has local predicates, the traditional way to calculate result cardinality is to multiply all the individual selectivities [68]. However, as it will be described in section 3.3.1, we use a more effective approach for this calculation.

Statistics Types: To measure the selectivity of a dataset for a specific range of values, we use quantile sketches. Following the Greenwald-Khanna algorithm [47], we extract quantiles which represent the right border of a bucket in an equi-height histogram. The buckets help us identify estimates for different ranges which are very useful in the case that filters exist in the base datasets. To find the number of unique values needed for formula 2.1, we use Hyperloglog [43] sketches. The HLL algorithm can identify with great precision the unique elements in a stream of data. We collect these types of statistics for every field of a dataset that may participate in any query. It should be noted that the gathering of these two statistical types happens in parallel.

Although AsterixDB has been enhanced with the aforementioned statistical types [21], its optimizer is not yet using them, leading to the following limitations:

- There is no selectivity estimation for predicates. Consequently, opportunities are missed for choosing the right join orders and join algorithms. Broadcast joins, in particular, will not be considered without a hint, even in the case when a dataset becomes small enough to fit in memory after the application of a selective filter.
- There is no cost-based join enumeration. Thus, a query's performance relies largely on the way it has been written by the user (i.e., the dataset ordering in the FROM clause).

Note that the above limitations are present in other existing large scale data platforms as well. We expect that the techniques presented in Chapter 3 would also be beneficial for those systems.

2.3 JSONiq features

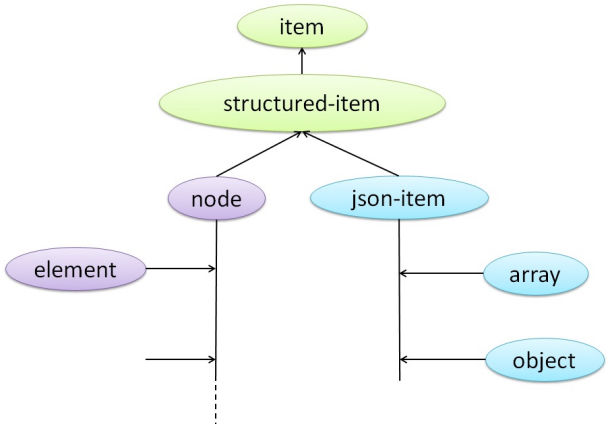


Figure 2.2: XML vs JSON structure

It is imperative for understanding the discussion in Chapter 4 to describe the representation along with the navigation expressions of JSON items according to the JSONiq extension to the XQuery specification. A json-item can be either an array or an object, in contrast to an XML structure, which consists of multiple nodes as described in Figure 2.2. An array consists of an ordered list of items (members), while an object consists of a set of pairs. Each pair is represented by a key and a value. The following is the terminology used for JSONiq navigation expressions:

- **Value:** for an array it yields the value of a specified (by an index) array element, while for an object it yields the value of a specified (by a field name) key.
- **Keys-or-members:** for an array it outputs all of its elements, and for an object it outputs all of its keys.

Chapter 3

Revisiting Runtime Dynamic Optimization for Join Queries in Big Data Management Systems

3.1 Introduction

Query optimization is a core component in traditional database systems, as it facilitates the order of execution decisions between query operators along with each operator's physical implementation algorithm. One of the most demanding operators is the Join, as it can be implemented in many different ways depending on the sizes of its inputs and outputs. To tackle the join optimization problem, two different approaches have been introduced.

The first approach (introduced in System R [26]) is cost-based query optimization; it performs an exhaustive search (through dynamic programming) among all different join

orderings until the one with the smallest cost is found and eventually executed in a pipelined mode. The second approach (introduced around the same time in INGRES [74]) uses instead a runtime dynamic query optimization method (later known as Adaptive Query Processing (AQP)), where the original query is decomposed into single-variable (i.e., single dataset) subqueries which are executed separately. This decomposition takes place in the following ways: (1) breaking off components of the query which are joined to it by a single variable, (2) substituting for one of the variables a tuple-at-a-time (to perform the join operation). Each subquery result is stored as a new relation that is then considered by the optimizer so as to optimize the remaining query. The choice of the "next" subquery to be executed is based on the cardinality of the participating datasets.

The INGRES approach was a greedy cardinality-based technique, with runtime overhead due to creating indexed (for joins) intermediate results, and the more comprehensive, cost-based, compile-time approach of System-R became the field's preferred approach [57, 69, 44, 46] for many years. To assign a cost for each plan (and thus find the best join ordering and implementation algorithms among the search space) the cost-based approach depends heavily on statistical information. The accuracy of such statistics is greatly affected by the existence of multiple selection predicates (on a single dataset), *complex* selection predicates (i.e., with parameterized values or UDFs) and join conditions that are not based on key-foreign key relationships. In such cases, statistics can be very misleading, resulting in inaccurate join result estimations. As the number of joins increases, the error can get worse as it gets propagated to future join stages [51]. These issues are exacerbated in today's big data management systems (BDMS) by the sheer volume of data.

In this work, we revisit the runtime dynamic optimization introduced by INGRES [74] and adapt it (with modifications) to a shared-nothing distributed BDMS, namely, AsterixDB. With the increase in the volume of data, even small errors in the join order can generate *very expensive* execution plans. A characteristic of the original dynamic optimization approach is that the choice of the "next" subquery to be executed is based only on dataset cardinality. However, the alternative cost-based optimization approach has shown that, for better join result estimation, one needs better statistics. Thus, we take advantage here of the materialization stages to collect all needed statistics. This combination of re-optimization and statistics collection leads to superior execution plans.

Specifically, when a query is executed, all predicates local to a table are pushed down and they are executed first to gather updated accurate statistics. The intermediate results along with the updated statistics are fed back to the optimizer to choose the cheapest initial join to be executed. This process is repeated until only two joins are left in the query. We integrated our techniques in AsterixDB [17, 24] which, like many relational database systems, is optimized for executing queries in a pipelined manner. Although with our modified dynamic optimization approach the query execution goes through blocking re-optimization points, this extra overhead is relatively minimal and is thus worthwhile since very expensive query plans are avoided.

Various works have been proposed in literature that use dynamic optimization techniques to alleviate the problems introduced by static cost-based optimization [28, 40, 52, 27, 70]. In this context, new statistics are estimated after mid-query execution (with information gathered from intermediate results) and they are used to re-calibrate the query

plan. This is similar to our approach; however, such works tend to ignore information coming from correlated selectivities, predicates with parameterized values and UDFs. Instead, by executing the local predicates first, we gain accurate cardinality estimations early that lead to improved query performance (despite the overhead of materializing those filters). Dynamic optimization has also been introduced in multi-node environments [59, 53, 23]. These works either introduce unnecessary additional overheads by running extra queries to acquire statistical data for the datasets [53] or they need to re-partition data because of lazily picking an inaccurate initial query plan [59]. Optimus [54] also uses runtime dynamic optimization, but it does not consider queries with multiple joins. Re-optimization points are used in [23] in a different way, as a place where an execution plan can be stopped if its execution is not as expected.

As we show in the experimental evaluation, for a variety of workloads, our modified runtime dynamic optimization will generate query plans that are better than even the best plans formed by (i) a user-specified order of the datasets in the FROM clause of a submitted query, or (ii) traditional static cost-based optimizers. In particular, our methods prevent the execution of expensive plans and promote more efficient ones. Re-optimizing the query in the middle of its execution and not focusing only on the initial plan can be very beneficial, as in many cases, the first (static) plan is changed dramatically by our optimizer.

In summary, this chapter makes the following contributions:

- We adapt an INGRES-like dynamic optimization scheme in a shared-nothing BDMS (AsterixDB). This includes a predicate pre-processing step that accurately estimates initial selectivities by executing all predicates local to a dataset early on. We insert multiple re-optimization points during query execution to receive feedback (updated

statistics for join results) and refine the remaining query execution plan. At each stage (i.e. re-optimization point), we only consider the next cheapest join, thus avoiding forming the whole plan and searching among all the possible join ordering variations.

- We assess the proposed dynamic optimization approach via detailed experiments that showcase its superiority against traditional optimizers. We also evaluate the overhead introduced by the multiple re-optimization points and the materialization of intermediate results.

The rest of this chapter is organized as follows: Section 4.2 discusses existing work on runtime dynamic optimization. Section 4.3 describes the details of the dynamic optimization approach including the use of statistics, while Section 3.4 showcases how the approach has been integrated into the current version of AsterixDB. The experimental evaluation appears in Section 3.5. Section 3.6 concludes this chapter and presents directions for future research.

3.2 Related Work

Traditional query optimization focuses on cost models derived from statistics on base datasets (cost-based optimization) as introduced in System R [26]. Typically, there are two steps in this process: first, there is a rewrite phase that transforms the specified query into a collection of alternate plans (created by applying a collection of rules), and second, cost models based on cardinality estimation are used to pick the plan with the least cost [44, 45, 39]. A cost-based optimization approach adapted for parallel shared-nothing architectures is described in [72]; here the master node sends the query to all worker nodes along with statistics. Then, each worker decides the best plan based on its restrictions

and sends its decision to the master. Finally, the master decides the globally optimal plan. This way, all the nodes in the cluster are working in parallel to find the best plan, each node working with a smaller set of plans. Our work also considers the shared-nothing environment, however, we concentrate on runtime dynamic optimization.

Runtime dynamic optimization was introduced in INGRES [74], where a query is decomposed into single-variable queries (one dataset in the FROM clause) which are executed separately. Based on the updated intermediate data cardinalities, the next best query is chosen for execution. In our work, we wanted to revisit this approach and see whether big data processing systems can benefit from it. Hence we execute part of the query to obtain statistics from the intermediate results and refine the remaining query. Opposite to INGRES, we do not depend only on cardinalities to build our cost model, but we collect more information regarding base and intermediate data based on statistics. Since INGRES, there have been various works using runtime dynamic optimization in a single-server context. Specifically, LEO [70] calibrates the original statistics according to the feedback acquired from historical queries and uses them to optimize future queries. In Eddies [27] the selectivity of each query operator is calculated while records are being processed. Eventually, the more selective operators are prioritized in the evaluation order.

Dynamic optimization is more challenging in a shared-nothing environment, as data is kept and processed across multiple nodes. Optimus [54] leverages runtime statistics to rewrite its execution plans. Although it performs a number of optimizations, it does not address multi-way joins, which as [54] points out, can be "tricky" because the data may need to be partitioned in multiple ways.

RoPE [23] leverages historical statistics from prior plan executions in order to tune future executions, e.g. the number of reduce tasks to schedule, choosing appropriate operations, including order. Follow-up work [33] extends the RoPE design to support general query workloads in Scope [80]. Their strategy generates a (complete) initial query plan from historical statistics, and it collects fresh statistics (specifically, partitioned histograms) during execution that can be used to make optimized adjustments to the remaining operators in the plan. However, in order not to throw away work, reoptimization takes place after a certain threshold and the initial plan is configured only based on the base datasets, which can potentially lead to suboptimal plans. In contrast, in our approach we block the query after each join stage has been completed and we use the result to optimize the subsequent stages; hence no join work is wasted. Furthermore, we estimate the selectivity of predicates by pushing down their execution; hence we avoid initial possibly misleading calculations. Nevertheless, learning from past query executions is an orthogonal approach that could be used to further optimize our approach and it is part of our future work.

Another approach belonging to the runtime dynamic optimization category uses *pilot runs*, as introduced in [53]. In an effort to alleviate the need for historical statistics, pilot runs of the query are used on sample data. There are two main differences between this approach and our work. First, statistics obtained by pilot runs are not very accurate for joins that do not have a primary/foreign key condition as sampling can be skewed under those settings. In contrast, our work gathers statistics on the base datasets which leads to more accurate join result estimations for those joins. Secondly, in our work we exploit AsterixDB's LSM ingestion process to get initial statistics for base datasets along with materialization of

intermediate results to get more accurate estimations - thereby we avoid the extra overhead of pilot runs.

Finally, RIOS [59] is another system that promotes runtime incremental optimization. In contrast to Optimus, RIOS assumes that the potential re-partitioning overhead is amortized by the efficiency of their approach. Particularly, statistics are collected during a pre-partitioning stage in which all the datasets participating in the query are partitioned according to an initial lazy plan formed based on raw byte size. However, if later statistics (collected during the pre-partitioning stage) indicate that this is not the correct plan, RIOS re-partitions the data. This is done if and only if the difference between the lazy plan and the better one is larger than a certain threshold. In that case, the remaining query is optimized according to the feedback acquired by intermediate results. In contrast to RIOS, our method alleviates the need for potential expensive re-partitioning since accurate statistics are collected before the query is processed by the optimizer. That way, we can pick the right join order from the beginning and thereby the right partitioning scheme. Hence, we avoid the overhead of faulty partitioning, which for large volumes can be very significant.

3.3 Runtime Dynamic Optimization

The main focus of our dynamic optimization approach is to utilize the collected statistics from intermediate results in order to refine the plan on each subsequent stage of a multi join query. To achieve this aim, there are several stages that need to be considered.

As described in Algorithm 1 lines 6-9, the first step is to identify all the datasets with predicates. If the number of predicates is more than one, or, there is at least one

complex predicate (with a UDF or parameterized values), we execute them as described in Section 3.3.1. Afterwards, while the updated query execution starts as it would normally do, we introduce a loop which will complete only when there are only two joins left in the query. In that case, there is no reason to re-optimize the query as there is only one possible remaining join order. This loop can be summarized in the following steps:

- A query string, along with statistics, are given to the **Planner** (line 12) which is responsible for figuring out the next best join to be executed (the one that results in the least cardinality) based on the initial or online statistics. As a result, the Planner does not need to form the complete plan, but only to find the cheapest next join for each iteration.

- The output plan is given as input to the **Job Construction** phase (line 14) which actually converts it to a job (i.e. creation of query operators along with their connections). This job is executed and the materialized results will be rewired as input whenever they are needed by subsequent join stages.

- Finally, if the remaining number of datasets is more than three, we return to the **Planner** phase with the new query as formatted in the **Query Reconstruction** phase (line 13); otherwise the result is returned.

3.3.1 Selective Predicates

Filtering can be introduced in the WHERE clause of a query in several forms; here we are focusing on selection predicates. In the case that a dataset has only one local selection

Algorithm 1 Dynamic Optimization Part 1

$J \leftarrow$ joins participating in the original query

$D \leftarrow$ collection of base datasets (d) in the query

$Statistics \leftarrow$ quantile and hyperloglog sketches for each field of D that is a join key

$Q(\sigma, D, J) \leftarrow$ original query as submitted by user ▷ σ is the projection list

for d in D **do**

    $P \leftarrow$ set of selective predicates local to d

    if $|P| > 1$ **then**

        $D - \{d\} \bigcup \text{PushDownPredicates}(d, P)$

    end if

end for

while $|J| > 2$ **do**

    $j \leftarrow \text{Planner}(\text{J, Statistics})$

    $Q(\sigma, D, J) \leftarrow \text{QueryReconstruction}(j, Q(\sigma, D, J))$

    $intermediateResults, Statistics \leftarrow ConstructAndExecute(j)$ ▷ collect statistical sketches on intermediate data and integrate them on the statistics collection framework

    $J \leftarrow$ joins in Q(D)

end while

$j \leftarrow \text{Planner}(J, Statistics)$

return ConstructAndExecute(j)

function PushDownPredicates(d, P)

    $Q(\sigma, \{d\}, \emptyset) \leftarrow$ query consists only of d with its local predicates ▷ σ is filled by fields participating in joins

Algorithm 2 Dynamic Optimization Part 2

$d', Statistics \leftarrow$ Execute(Q$(\sigma, \{d\}, \emptyset)$) ▷ update original Statistics with the sketches collected for the new d

return d'

end function

function PLANNER$(J, Statistics)$

$minJoin \leftarrow \emptyset$, $finalJoin \leftarrow \emptyset$

for j in J **do**

minJoin$\leftarrow$ min(minJoin, JoinCardinality$(j, Statistics)$)

end for

if $|J| = 2$ **then**

$finalJoin \leftarrow BestAlgorithm(minJoin) \bowtie BestAlgorithm((J - \{minJoin\}))$

else

$finalJoin \leftarrow BestAlgorithm(minJoin)$

end if

return $finalJoin$

end function

function QUERYRECONSTRUCTION$(j(d_1, d_2), Q(\sigma, D, J))$

$d' \leftarrow CreateDataset(j(d_1, d_2))$

$D \leftarrow (D \bigcup \{d'\}) - \{d_1, d_2\}$

$J \leftarrow J - \{j(d_1, d_2)\}$

return $Q(\sigma, D, J)$

end function

predicate with fixed value, we exploit the equi-height histogram's benefits. Particularly, depending on the number of buckets that we have predefined for the histogram, the range cardinality estimation can reach high accuracy.

However, for multiple selection predicates or complex predicate(s), the prediction can be very misleading. In the case of multiple (fixed value) predicates, traditional optimizers assume predicate independence and thus the total selectivity is computed by multiplying the individual ones. This approach can easily lead to inaccurate estimations [50]. In the absence of values for parameters, and given non-uniformly distributed data (which is the norm in real life), an optimizer cannot make any sort of intelligent prediction of selectivity, thus default values are used as described in [68] (e.g. 1/10 for equalities and 1/3 for inequalities). The same approach is taken for predicates with UDFs [49]. Most works dealing with complex predicates [38, 49] focus on placing such predicates in the right order and position within the plan, given that the selectivity of the predicate is provided. In our work, we exploit the INGRES [74] approach and we push down the execution of predicates (lines 20-23 of Algorithm 2) to acquire accurate cardinalities of the influenced datasets.

As a complex predicate example consider the following query Q_1, where we have four datasets, two of which are filtered with UDFs and then joined with the remaining two. (For simplicity in this example we use UDFs but the same procedure is followed for predicates with parameterized values.)

```
select A.a
from A, B, C, D
where udf(A) and A.b = B.b
```

```
and udf(C) and B.c = C.c
and B.d = D.d;
```

As indicated in line 21 of Algorithm 2, we isolate the datasets enhanced with local filters and we create queries for each one of those similarly to the decomposition technique used in INGRES to create single variable queries. In Q_1, datasets A and C will be wrapped around the following single variable queries (Q_2 and Q_3 accordingly):

```
select A.a, A.b
from A
where udf(A);
```

```
select C.c
from C
where udf(C);
```

Note that in both queries the SELECT clause is defined by attributes that participate in the remaining query (i.e in the projection list, in join predicates, or in any other clause of the main query). Once the query construction is completed, we execute them and we save the intermediate results for future processing from the remaining query. At the same time, we also update the statistics (hyperloglog and quantile sketches) attached to the base unfiltered datasets to depict the new cardinalities. Once this process is finished, we need to update Q_1 with the filtered datasets (line 9 in Algorithm 1), meaning removing the UDFs and changing the FROM clause. The final query which will be the input to the looping part of our algorithm (lines 11-18) is illustrated below as Q'_1.

```
select A'.a
from A', B, C', D
where A'.b = B.b and B.c = C'.c
and C'.d = D.d;
```

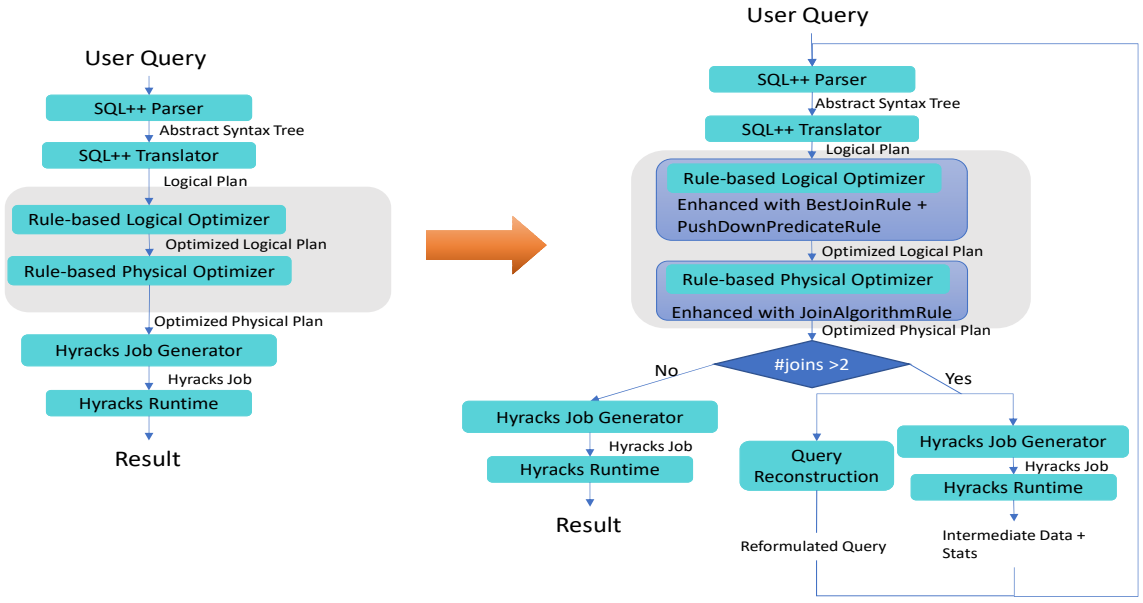


Figure 3.1: AsterixDB workflow without and with the integration of Dynamic Optimization

3.3.2 Planner

Next is the Planner stage (lines 25-30), where the input is the non-optimized query (in our case Q_1'), along with the most updated statistics. The goal of this stage is to output the best plan (since we focus on joins, this is the plan containing the best join order and join algorithm).

The first step in the Planner phase is to identify the join with the least result cardinality, along with its algorithm (lines 27-28). After that, we need to construct the join which will be output. If there are more than two joins in the input, then the cheapest join is the output and we are done (lines 31-32). However, in the case that there are only two joins, the Planner will pick the most suitable algorithm for both joins. Then, it will combine

the two joins by ordering them according to their result cardinality estimation (lines 29-30 of Algorithm 2).

In Q_1' there are three joins, which means that the first case is applied and it suffices to find the cheapest join according to statistics. Assuming that according to formula 2.1, A' and B lead to the smallest result cardinality, and A' (after the UDF application) is small enough to be broadcast, the plan output is a broadcast algorithm between A' and B ($J_{A'B}$).

3.3.3 Job Construction

Next, we construct a job for the plan (in our example, $J_{A'B}$) output by the previous stage (lines 14 and 18 of Algorithm 1). The details of how we construct a job in AsterixDB are described in section 3.4.3. The way a job is executed depends on the number of joins in the plan. If there is only one join, it means that we are still inside the looping part of the algorithm (line 14). To that end, we need to materialize the intermediate results of the job and at the same time gather statistics for them. In our example, plan $J_{A'B}$ has only one join - thereby the aforementioned procedure will be followed and the joined results of A' and B will be saved for future processing along with their statistics.

On the other hand, if the plan consists of two joins, it means that the dynamic optimization algorithm has been completed and the results of the job executed are returned back to the user (line 18 of Algorithm 1).

Online Statistics: For the statistics acquired by intermediate results, we use the same type of statistics as described in section 2.2. We only gather statistics on attributes that participate on subsequent join stages (and thus avoid collecting unnecessary information). The online statistics framework is enabled in all the iterations except for the last one (i.e.

the number of remaining datasets is three) since we know that we are not going to further re-optimize.

3.3.4 Query Reconstruction

The final step of the iterative approach is the reconstruction of the remaining query (line 13 of Algorithm 1). Given that there will be more re-optimization points (more than two joins remaining), we need to reformulate the remaining query since the part that participates in the job to be executed needs to be removed. The following issues need to be considered in this stage:

- The datasets participating in the output plan need to be removed (as they are not going to participate in the query anymore) and replaced by the intermediate joined result (lines 36-37).
- The join output by Planner needs to be removed (line 38).
- Any other clause of the original query influenced by the results of the job just constructed, needs to be reconstructed.

Following our example, the Planner has picked as optimal the join between A' and B datasets. Consequently this join is executed first; then, the joined result is stored for further processing and is represented by a new dataset that we call I_{AB}. In terms of the initial query, this will trigger changes in all its clauses. Particularly, in the select clause the projected column derives from one of the datasets participated in the subjob (A). Hence, after its execution, the projected column will now derive from the newly created dataset I_{AB}. In the FROM clause both A and B should be removed and replaced by I_{AB}. Finally,

in the WHERE clause, the join executed has to be removed and if its result participates in any of the subsequent joins, a suitable adjustment has to be made. To this end, in our example B is joined with C in its c attribute. However, the c column is now part of I_{AB}. As a result, I_{AB} will now be joined with C. After these changes the reformatted query will look like this (Q_4):

```
select I_AB.a
from I_AB, C, D
where I_AB.c = C.c and C.d = D.d;
```

Q_4 has only two joins, which means that the looping part of our algorithm has been completed and that once the Planner picks the optimal join order and algorithm the final job will be constructed and executed with its results returned to the user.

3.3.5 Discussion

By integrating multiple re-optimization points during mid-query execution and allowing complex predicate pre-processing, our dynamic optimization approach can lead to much more accurate statistics and efficient query plans. Nevertheless, stopping the query before each re-optimization point and gathering online statistics to refine the remaining plan introduces some overhead. As we will see in the experimental section, this overhead is not significant and the benefits brought by the dynamic approach (i.e., avoiding a bad plan) exceed it by far. Note that here we focus on simple UDF predicates applied on the base datasets. For more expensive UDF predicates, plans that pull up their evaluation need to be considered [49]. Another interesting point unlocked by dynamic optimization is the forming

of bushy join plans. Although they are considered to be expensive as both inputs of the join need to be constructed before the join begins in a parallel environment, they tend to be very efficient as they can open opportunities for smaller intermediate join results.

3.4 Integration into AsterixDB

As AsterixDB is supported by two other frameworks (Algebricks and Hyracks), there were multiple changes needed so as to integrate the dynamic optimization approach. The left side of Figure 3.1 represents the current query processing workflow of the AsterixDB framework, while the right side summarizes our changes. In particular, in the beginning the workflow behaves in the same way as always, with the exception of few additional rules integrated into the rule-based (JoinReOrderRule, PushDownPredicateRule) and physical-based (JoinAlgorithmRule) optimizer (**Planner**). Afterwards, depending on the number of joins participating in the query currently being processed, we either construct and execute the Hyracks job and output the result to the user as usual (only two joins) or we perform the following two steps (more than two joins):

- We introduce the **Query Reconstruction** phase where we reformulate the query currently being processed and we redirect it as new input to the SQL++ parser and the whole query process starts from the beginning once again.
- We construct a Hyracks job (**Job Construction**) by using various new operators introduced to allow materialization of the results of the query currently being processed along with connection of previously (if any) executed jobs.

3.4.1 Planner

If a dataset has more than one filter, the PushDownPredicateRule is triggered. This rule will push the filters down to their datasource and will remove the rest of the operators from the plan, leading to a modified plan of a simple select-project query (like Q_2 and Q_3 in section 3.3.1) . On the other hand, if there is only one filter, we estimate the filtered dataset cardinality based on histograms built on the base dataset.

Afterwards, the Planner stage will decide the optimal join order and algorithm. In order for the Planner to pick the join with the least cardinality, we enhanced the rule-based logical Optimizer (part of the Algebricks framework) with the JoinReOrderRule (see Figure 3.1). To further improve the efficiency of the execution plan, we integrated a rule in the rule-based physical Optimizer (Figure 3.1) that picks the most suitable join algorithm.

Join Ordering

The main goal of the join order rule is to figure out the join with the least cardinality. To that end, we identify all the individual joins along with the datasources (post-predicate execution) of their predicates. In this work, we focus only on joins as formed in the WHERE clause of the query. In the future, we plan to infer more possible joins according to correlations between join predicates. Afterwards, we apply formula 2.1 based on statistics (see Section 2.2) collected for the datasets and predicates involved in the join.

Traditional optimizers that are based on static cost-based optimization need to form the complete plan from the beginning, meaning that we need to search among all different possible combinations of joins which can be very expensive depending on the number

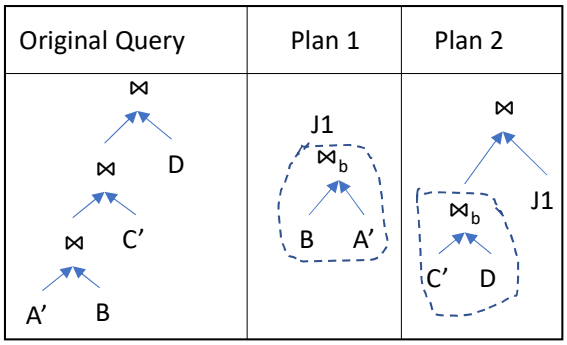


Figure 3.2: Planning Phase when Dynamic Optimization is triggered

of base datasets. However, in the case of incremental optimization, it suffices to search for the cheapest join because the rest will be taken into consideration in the next iterations of our algorithm. In our example in Figure 3.2, in Q_1 the join between post-predicate A (A') and B will be estimated as the cheapest one and will be output from the Planner stage.

The second feature of this rule is triggered when there are only two joins left in the query and hence the statistics obtained up to that point suffice to figure out the best join order between them. Specifically as depicted in Plan 2 of Figure 3.2, in this case a two-way join (between three datasets) is constructed whose inputs are (1) the join (between two of the three datasets) with the least result size (estimated as described above) and (2) the remaining dataset.

It is worth noticing that in the first iteration of the approach the datasets that are joined are always among the base datasets. However, in the rest of the iterations, one or both of the joined datasets may be among the results from previous iterations. An example of that is shown in Plan 2 of Figure 3.2, where the right dataset of the final join is the result of the first iteration (J1) of our algorithm.

Join Algorithm

While hash join is the default algorithm, by having accurate information about the datasets participating in the corresponding join, the optimizer can make more efficient decisions. If one of the datasets is small enough, like A' and C' in our example (see Figure 3.2), then it can be faster to broadcast the whole dataset and avoid potential reshuffling of a large dataset over the network.

Knowing that the cardinality of one of the datasets is small enough to be broadcast also opens opportunities for performing the indexed nested loop join algorithm as well. However, two more conditions are necessary to trigger this join algorithm. The first one is the presence of a secondary index on the join predicate of the "probe" side. The second condition refers to the case of primary/foreign key join and dictates that the dataset that gets broadcast must be filtered - thereby during the index lookup of a large dataset there will be no need for all the pages to be accessed.

3.4.2 Query Reconstruction

This stage is entered in one of the following cases: (1) the Planner has output a simple projection plan (predicate push down) or (2) the Planner output is a select-project-join plan (cheapest join). In both cases, we follow the process described in section 3.3.4 to reformulate the clauses of the input query and output the new query that will be given as input to the optimizer for the remaining iterations of our algorithm.

3.4.3 Job Construction

There are three different settings when creating a job:

1. When there are still re-optimizations to be scheduled (more than 2 joins), the output of the job has to be materialized for future use.

2. If one or both inputs of a job is a previously materialized job output, we need to form a connection between the jobs.

3. When the iterations are completed, the result of the last job will be returned to the user.

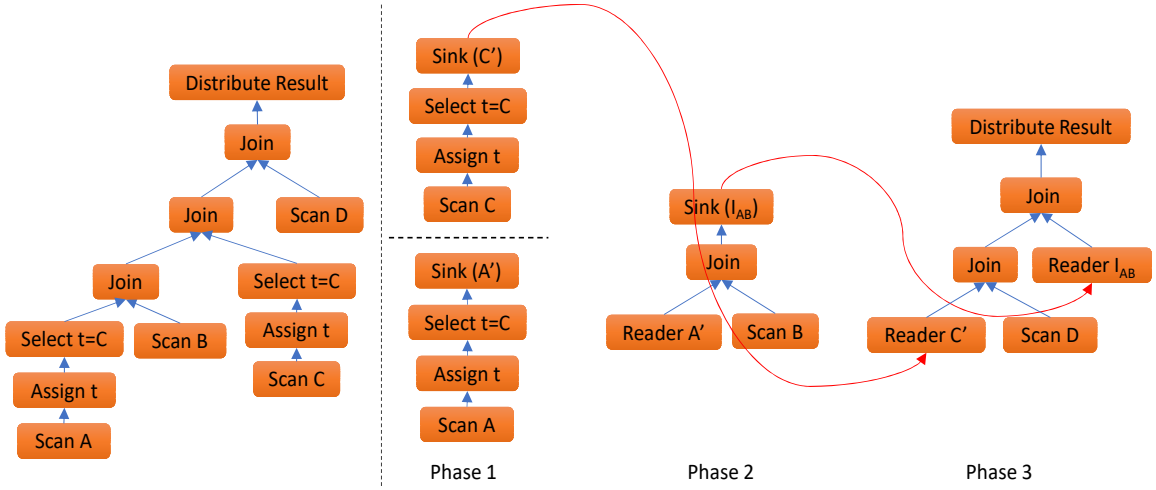


Figure 3.3: Original Hyracks job split into smaller jobs

We use the example in Figure 3.3 to illustrate the process we followed to satisfy the aforementioned cases. The left side of the figure depicts the usual job for the three-way join query (Q_1), where the final result is returned to the user via the DistributeResult operator. Instead, on the right side of the Figure (Phase 1), two subjobs are created which push down the UDF predicates applied to datasources A and C. Their results are the post-predicate versions of A and C (Sink(A') and Sink(C') accordingly). The Sink operator is responsible for materializing intermediate data while also gathering statistics on them.

In Phase 2, the subjob formed wraps the join between datasets A' and B, as this is the plan output by the Planner. Note that the new operator introduced in this phase (Reader A') indicates that a datasource is not a base dataset. Instead, it is intermediate data created by a previous subjob. In our example, Reader A' represents the materialized data created in the previous phase by Sink(A'). Since the original query has not finished yet (remaining joins), the Sink operator will be triggered once again and it will store in a temporary file the joined results (I_{AB}), while at the same time it will collect the corresponding statistics.

Finally, the goal of Phase 3 is to wrap the output of the Planner which is a two-way join. The existence of two joins indicates that we are at the final iteration of the dynamic approach - thereby this job is the final one and its result should be returned to the user. Consequently, the DistributeResult operator re-appears in the job, as depicted in Figure 3.3.

3.4.4 Discussion

To integrate the dynamic optimization approach in the AsterixDB framework, we had to create an iterative workflow which gave us the opportunity to trigger multiple re-optimization points that result in more efficient query plans. In this work, we concentrate on multi-join queries which may also contain multiple and/or complex selection predicates. Although other types of operators may exist in the query, for now they are evaluated after all the joins and selections have been completed and traditional optimization has been applied. In the future, we plan to investigate more costly UDF predicates that may instead be better to be pulled up for evaluation.

Query 17: $(\sigma_{m,y}\,(\text{date\_dim}))\bowtie_{dsk=rdsk}\text{store\_returns}\bowtie_{i=i,tn=tn,c=c}\text{store\_sales}\bowtie_{s=s}\text{store}\bowtie_{sdsk=dsk}(\sigma_{m,y}\,(\text{date\_dim}))\bowtie_{i=i}\text{item}\bowtie_{c=bc,i=i}$ catalog_sales $\bowtie_{sdsk=dsk}(\sigma_{m,y}\,(\text{date\_dim}))$

Query 50: $(\sigma_{y,m}\,(\text{date\_dim}))\bowtie_{dsk=rdsk}\text{store\_returns}\bowtie_{i=i,tn=tn,c=c}\text{store\_sales}\bowtie_{sdsk=dsk}\text{date\_dim}\bowtie_{s=s}\text{store}$, with y=rand(1998,200), m=rand(8,10)

Query 9: $(\sigma_b\,(\text{part}))_{k=k}\bowtie\text{lineitem}\bowtie_{sk=sk}\text{supplier}\bowtie_{nk=nk}\text{nation}\bowtie_{sk=sk,pk=pk}\text{part\_sup}\bowtie_{ok=ok}(\sigma_{d,p}\,(\text{order}))$, with σ_b =(mysub(b)="#3"), σ_d= (myyear(d)=1998)

Query 8: $\text{lineitem}\bowtie_{pk=pk}(\sigma_t\,(\text{part}))\bowtie_{sk=sk}\text{supplier}\bowtie_{ok=ok}(\sigma_{d,s}\,(\text{order}))\bowtie_{ck=ck}\text{customer}\bowtie_{nk=nk}\text{nation}\bowtie_{rk=rk}(\sigma_n\,(\text{region}))\bowtie_{nk=nk}\text{nation}$

Figure 3.4: Queries used for the experimental comparisons.

3.5 Experimental Evaluation

We proceed with the performance evaluation of our proposed strategies and discuss the related trade-offs. The goals of our experiments are to: (1) evaluate the overheads associated with the materialize and aggregate statistics steps; (2) show that good join orders and methods can be accurately determined, and (3) exhibit the superior performance and accuracy over traditional optimizations. In particular, in the following experiments we compare the performance of our dynamic approach with: (i) AsterixDB with the worst-order, (ii) AsterixDB with the best-order (as submitted by the user), (iii) AsterixDB with static cost-based optimization, (iv) the pilot-run [53] approach, and (v) an INGRES-like approach [74]. Section 3.5.2 contains detailed explanations of each optimization approach.

Experimental Configuration: All experiments were carried out on a cluster of 10 AWS nodes, each with an Intel(R) Xeon(R) E5-2686 v4 @ 2.30GHz CPU (4cores), 16GB of RAM and 2TB SSD. The operating system is 64-bit Red-Hat 8.2.0. Every experiment was carried out five times and we calculated the average of the results.

```
SELECT ...
FROM store_sales, store_returns, catalog_sales, date_dim d1, date_dim
    d2, date_dim d3, store, item
WHERE d1.d_moy = 4
AND d1.d_year = 2001
AND d1.d_date_sk = ss_sold_date_sk
AND i_item_sk = ss_item_sk
AND s_store_sk = ss_store_sk
AND ss_customer_sk = sr_customer_sk
AND ss_item_sk = sr_item_sk
AND ss_ticket_number = sr_ticket_number
AND sr_returned_date_sk = d2.d_date_sk
AND d2.d_moy BETWEEN 4 AND 10
AND d2.d_year = 2001
AND sr_customer_sk = cs_bill_customer_sk
AND sr_item_sk = cs_item_sk
AND cs_sold_date_sk = d3.d_date_sk
AND d3.d_moy BETWEEN 4 AND 10
AND d3.d_year = 2001
GROUP BY i_item_id, i_item_desc, s_store_id, s_store_name
ORDER BY i_item_id, i_item_desc, s_store_id, s_store_name
LIMIT 100 ;
```

Listing 3.1: TPC-DS Query 17

Queries: We evaluate the performance using four representative queries from TPC-DS (Query 17 and Query 50) [19] and TPC-H [20] (Query 8 and Query 9). The

actual queries are shown in Figure 4.4.2. These queries were selected because of: (1) their complexity (from the number of joins perspective), and, (2) their variety in join conditions (primary/foreign key vs fact-to-fact joins).

To better assess the effect of selection predicates on our runtime dynamic approach, we used modified versions of Queries 8, 9 and 50. Specifically, to consider multiple fixed value predicates, in Query 8 we added two (and correlated [78]) predicates on the *orders* table. We use Query 9 to examine the effect of UDFs (by adding various UDFs on top of the *part* and *orders* tables. Finally, in Query 50, we added two selections with parameterized values on top of one of the dimension tables.

```
SELECT ...
 FROM store_sales, store_returns, date_dim d1, date_dim d2, store
 WHERE d1.d_moy = myrand(8,10)
 AND d1.d_year = myrand(1998,2000)
 AND d1.d_date_sk = sr_returned_date_sk
 AND ss_customer_sk = sr_customer_sk
 AND ss_item_sk = sr_item_sk
 AND ss_ticket_number = sr_ticket_number
 AND ss_sold_date_sk = d2.d_date_sk
 AND ss_store_sk=s_store_sk ;
```

Listing 3.2: TPC-DS Query 50

For all of the scenarios we generate 3 TPC-DS and 3 TPC-H datasets with scale factors 10, 100, 1000. A scale factor of 1000 means that the cumulative size for the datasets involved in the specific query is 1TB. All the data is directly generated and then loaded

into AsterixDB. It is also worth noting that we gain upfront statistics for the forming of the initial plan during the loading of the datasets in AsterixDB. This is only performed once and it is not part of the query execution process; thus the performance numbers reported in our results do not include that part. The loading times can vary from 10 minutes to 8 hours depending on the size of the datasets. However, as was shown in [21], the statistics collection overhead is minimal with respect to the loading time.

```
SELECT ...
FROM part, supplier, lineitem, orders, customer, nation n1, nation n2,
    region
WHERE p_partkey = l_partkey
AND s_suppkey = l_suppkey
AND l_orderkey = o_orderkey
AND o_custkey = c_custkey
AND c_nationkey = n1.n_nationkey
AND n1.n_regionkey = r_regionkey
AND r_name = 'ASIA'
AND s_nationkey = n2.n_nationkey
AND o_orderdate between date '1995-01-01' and date '1996-12-31'
AND o_orderstatus='F'
AND p_type ='SMALL PLATED COPPER';
```

Listing 3.3: TPCH Query 8

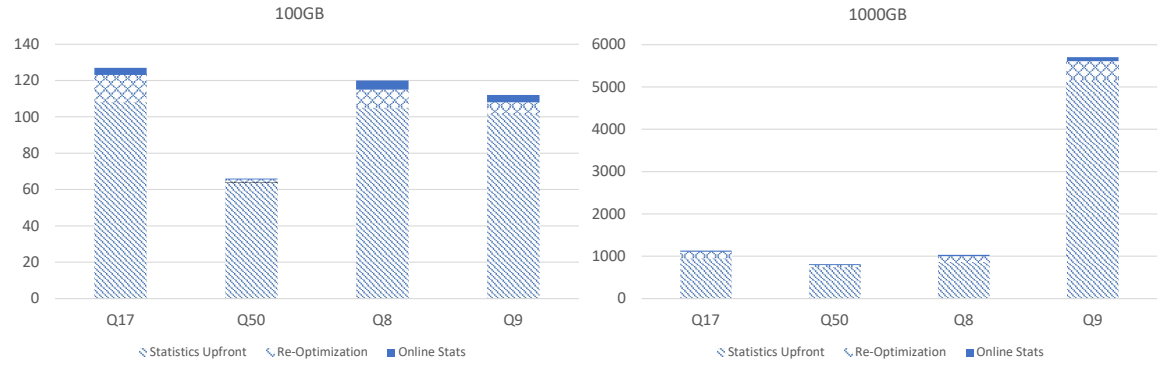
```
SELECT ...
FROM part, supplier, lineitem, partsupp, orders, nation
WHERE s_suppkey = l_suppkey
AND ps_suppkey = l_suppkey
AND ps_partkey = l_partkey
AND p_partkey = l_partkey
AND o_orderkey = l_orderkey
AND myyear(o_orderdate) = 1998
AND s_nationkey = n_nationkey
AND mysub(p_brand)='#3';
```

Listing 3.4: TPCH Query 9

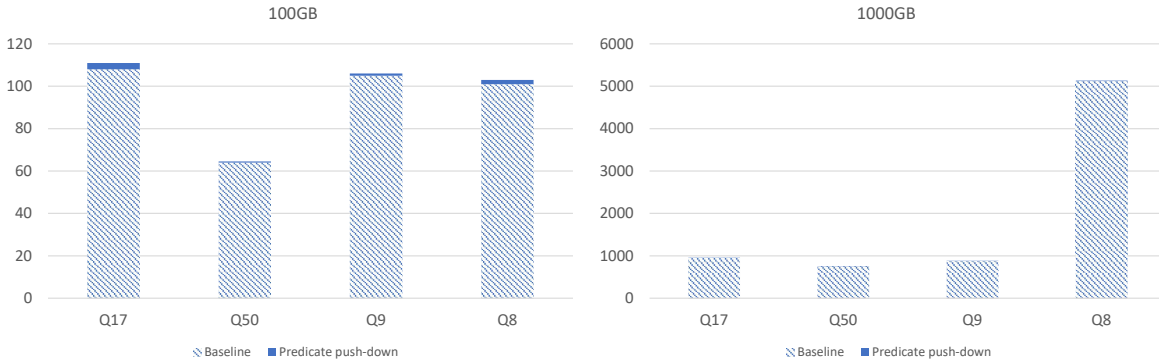
3.5.1 Overhead Considerations

In this section, we evaluate the overhead introduced to the AsterixDB execution time due to our dynamic optimization techniques, namely (1) the introduction of re-optimization points, (2) the gathering of statistics during runtime, and (3) the separate execution of multiple/complex predicates. To this end, we report the execution times for the above four representative queries for scale factors 100 and 1000.

For the first two settings we perform the following three executions for each query. In the first execution we acquired all the statistics needed for forming the optimal execution plan by running our runtime dynamic optimization technique. Then, we re-executed the query by having the updated statistics for each dataset so that the optimal plan is found from the beginning. In the final execution, we enabled the re-optimization points but we



(a)



(b)

Figure 3.5: Overhead imposed by (a) multiple re-optimization points and the online statistics and (b) pre-execution of complex predicates.

removed the online statistics collection. That helped us assess the overhead coming from writing and reading materialized data. Finally, to evaluate the cost of online statistics gathering we simply deducted the third execution time (re-optimization) from the first one (whole dynamic optimization technique).

As seen in figure 3.5(a), for scale factor 100, the total re-optimization time is around 10% of the execution time for most queries, with the exception of Q50 which has only four joins leading to an overhead of 2%. Particularly, the four joins introduce two re-optimization points before the remaining query has only two joins and there is no need for further re-

optimization. There is also a re-optimization in the beginning of this query introduced by the execution of the filtered dataset. However, this is insignificant as will be discussed later. For the scale factor of 1000, the overhead of re-optimization increases up to 15% for most queries, as the intermediate data produced are larger and thus the I/O cost introduced by reading and writing intermediate data is increased.

The online statistics collection brings a small overhead of 1% to 3% (scale factor 100) to the total execution time, as it is masked from the time we need to store and scan the intermediate data. Moreover, the extra time for statistics depends on the number of attributes for which we need to keep statistics for. Following the example of Q50 as above, the statistics collection overhead is only 1% because it has the smallest number of join conditions. In scale factor 1000 the overhead of gathering statistics is increased, as the data upon which we collect statistics are larger in size, but it remains insignificant (up to 5%). Overall, we observe a total of 7-13% overhead for scale factor 100 and up to 20% for scale factor 1000. We believe that this is acceptable given the benefits brought by our approach, as will be shown in Section 3.5.2.

Finally, we assess the overhead of applying the incremental optimization approach to estimate the influences of multiple/complex predicates. For the base setup, we deactivated the multiple re-optimization points and executed the plan formed as if the right statistical data is available from the beginning. Then, the experiment was repeated by enabling the dynamic optimization only for materializing the intermediate results coming from pushing down and executing multiple predicates. The remaining query was executed based on the refined statistics coming from the latter step. As the results show (see figure 3.5(b)), even

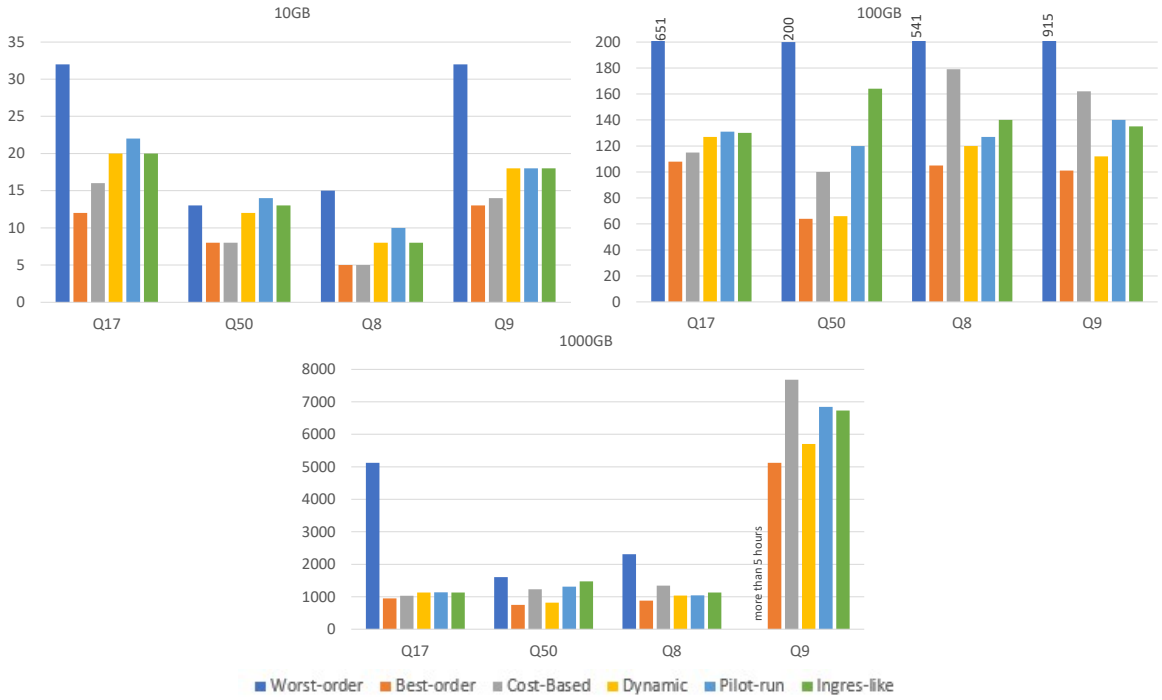


Figure 3.6: Comparison between Dynamic Optimization, traditional cost-based optimization, regular AsterixDB (join best-order vs worst-order), Pilot-run and Ingres-like.

in the case of Q17, where there are multiple filters present, the overhead does not exceed 3% of the total execution time, even for scale factor 1000. On the other hand, Q50 once again has the smallest overhead as there is only one dataset filtered.

3.5.2 Comparison of Execution Times

We proceed to evaluate our dynamic approach techniques against: (i) the join worst-order, (ii) the join best-order, (iii) a traditional cost-based optimization and (iv) the pilot-run method [53]. For the worst-order plan, we enforce a right-deep tree plan that schedules the joins in decreasing order of join result sizes(the size of the join results was computed during our optimization). The best-order plan assumes that the user knows the

optimal order generated by our approach and uses that order in the FROM clause when writing the query. We also put some broadcast hints so the default optimizer can choose the broadcast algorithm. These two settings represent the least and the most gain, accordingly, that we can achieve with our approach against the default approaches of AsterixDB.

To compare with a traditional cost-based optimization approach, we collected statistics on the base datasets during the ingestion phase and we formed the complete execution plan at the beginning based on the collected statistics. When UDFs or parameters are present in a query we use the default selectivity factors as described in [68]. For the pilot-run method, we gathered the initial statistics by running select-project queries (pilot-runs) on a sample of each of the base datasets participating in the submitted query. If there are predicates local to the datasets, they are included in the pilot-runs. In the sampling technique used in [53] during pilot runs, after k tuples have been output the job stops. To simulate that technique we enhanced our "pilot runs" with a LIMIT clause. Based on those statistics, an initial (complete) plan is formed and the execution of the original query begins until the next re-optimization point where the plan will be adjusted according to feedback acquired by online statistics.

Finally, for the INGRES-like approach we use the same approach as ours to decompose the initial query to single variable queries. However, the choice of the next best subquery to be executed is only based on dataset cardinalities (without other statistical information). Furthermore, in the original INGRES approach intermediate data are stored into a new relation; in our case we store it in a temporary file for simplicity. The experimental results are shown in Figure 3.6.

TPC-DS

Query 17: This query has a total of 8 base tables (Figure 4.4.2). Three of those (i.e. dimension tables) are attached to selective filters and are used to prune down the three large fact tables, while *item* and *store* (i.e. smaller tables) are used for the construction of the final result. Our dynamic optimization approach will find that the optimal plan is a bushy tree, as dimension tables should be joined with the fact tables to prune down as much as possible the intermediate data. Then, they will be joined with each other to form the result. It is also worth noting that our approach will find that the dimension tables and *store* will be broadcast in all scale factors along with *item* in factors 10 and 100.

Given that there are no complex predicates, all other approaches (apart from the worst-order) will form similar bushy trees along with the suitable join algorithm in the appropriate cases. Hence, our dynamic optimization approach does not bring any further benefit (in fact there is a slight degradation, around 1.15-1.20x depending on the scale factor, against best-order due to the overhead introduced by re-optimization). Finally, the worst-order will join the fact tables first, resulting in very large intermediate results and a 5x slower performance.

Query 50: This query contains two dimension tables (*date_dim*) only one of which is filtered (with parameterized expressions), two large tables and *Store* that helps pruning down the final result. The optimal plan found by our dynamic approach first prunes down one of the fact tables by joining it with the filtered dimension table and then joins it with the other large table. Our approach is also able to choose the broadcast algorithm whenever appropriate. With the enhancement of broadcast hints, best-order will pick exactly the

same execution plan, leading to slightly better performance than our dynamic approach (1.05, 1.1x for scale factors 100 and 1000).

Cost-based optimization results in a different plan because of the inaccurate cardinality estimates on the post-filtered dimension table and on the joined result between the fact tables. As a result, although it finds most of the broadcast joins, it leads to a 1.5x worse performance than our approach for scale factors 100 and 1000. A bushy tree will be formed by the INGRES-like approach due to its naive cost-model approach (considering only dataset cardinalities), resulting in an even worse performance. The worst-order of AsterixDB will trigger hash joins by default. On top of that, it will schedule the join between the fact tables in the beginning; thus it has the worst performance. Lastly, pilot-run makes the wrong decision concerning the join ordering between the large tables because of inaccurate statistics and thereby is around 1.8x slower than our approach.

TPC-H

Query 9: The lineitem table is joined on foreign/primary key with four smaller tables and on foreign key with *part_sup*. Once again, our approach will find the optimal plan, which in this case is a bushy tree. Apart from the correct join-order, our techniques will pick the broadcast algorithm in the case of the *part* table for scale factors 10 and 100, as well as in the case of the joined result of *nation* and *supplier* tables. Cost-based optimization will find a similar bushy tree; however, due to wrong cardinality estimation, it will not broadcast the part table and the intermediate data produced by joining nation and supplier will only be broadcast for scale factor 10. As a result, our approach has a slightly

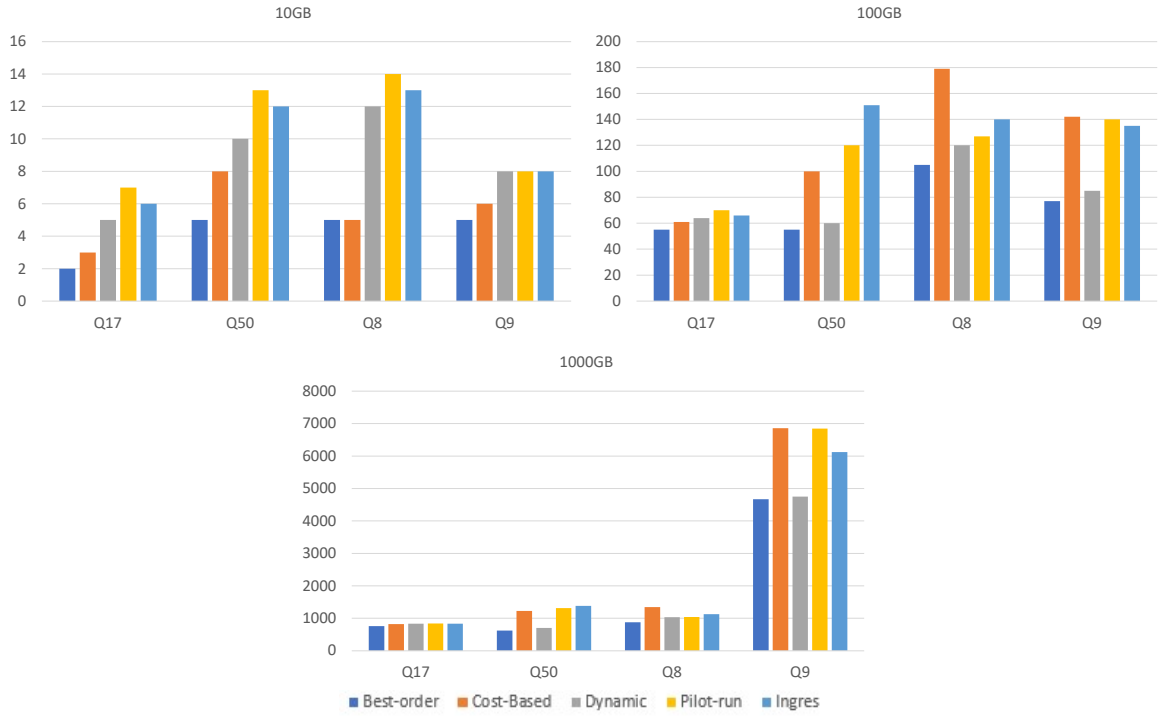


Figure 3.7: Comparison between Dynamic Optimization, traditional cost-based optimization, regular AsterixDB (join best-order vs worst-order), pilot-run and ingres-like when INL join is considered.

better performance than the cost-based one. Similarly, the best-order will form the optimal execution plan leading to the best performance once again.

As with all the other queries, the worst-order will schedule the largest result producing joins in the beginning along with the hash algorithm, which will result in an execution time more than 5 hours. Hence, almost all techniques were 7x better than the worst-order. In the pilot-run case, once again, a suboptimal plan is chosen due to inaccurate unique cardinalities estimated by initial sampling. Finally, once again the INGRES-like approach will form a less efficient bushy tree since it focuses only on dataset cardinalities.

Query 8: This query has eight datasets in total. The lineitem table is a large fact table while all the others are smaller (three of them are filtered with multiple predicates).

All the joins between the tables are between foreign/primary keys. Again our approach manages to find the optimal plan (bushy join) as it uses the dynamic optimization techniques described above to calculate the sizes of base datasets after multiple-predicate filters are applied. The dynamic approach also gives the opportunity to the optimizer to choose the broadcast algorithm when appropriate, mainly for scale factors 10 and 100. Best-order will form the same execution plan (both in terms of join order and algorithm) as the dynamic approach and it will be more efficient since there is no re-optimization.

In the cost-based case, due to inaccurately estimated cardinalities on the post-filtered *orders* table, a different bushy plan is chosen. Although for scale factor 1000, the benefit of broadcast opportunities picked by the dynamic approach is not as noticeable as in the rest of the scale factors, it is still 1.3x faster than the cost-based one since it forms a better plan. Furthermore, pilot-run forms the same optimal plan as our approach, but because of the overhead introduced by pilot runs is slightly slower. The INGRES-like approach will focus only on dataset cardinalities and not on statistical information and thus it will find a suboptimal plan. Finally, the worst-order leads to a right-deep join with hash joins that can be up to 2.5x worse than our approach.

3.5.3 Considering Indexed Nested Loop Join

The last set of experiments examine the behavior of our approach when the Indexed Nested loop Join (INLJ) is added as another possible join algorithm choice. We thus enhanced the TPC-H and TPC-DS datasets with a few secondary indexes on the attributes that participate in queries as join predicates and are not the primary keys of a dataset. The worst-order is excluded from these experiments since in the absence of hints, it will not

choose INL; hence its execution time will not change. The results of these experiments are shown in Figure 3.7.

TPC-DS

Query 17: In this particular query, there are 3 cases where the INL join will be picked by the dynamic approach for all scale factors. All of these cases are for the foreign/primary key joins between the large fact tables and the post-filtered dimension tables. In these particular cases the dimension tables are small enough to be broadcast but at the same time they have been filtered; hence not all pages of the large fact tables satisfy the join and need to be accessed. The same will happen with all the other approaches - thereby the execution time will be better in all cases. To that end, our dynamic approach will not bring any further benefit in this particular case.

Query 50: In this query, the dynamic approach will pick the INL join algorithm only in the case of the join between the filtered dimension table and the *store_returns* table. However, *store_returns* is not a very large table, and thus scanning it instead of performing an index lookup does not make a big difference; this results in a smaller improvement compared to the performance in the previous section. The INGRES-like approach similar to the dynamic one, will pick the INL join for *store_returns*$\bowtie$*date_dim* because *date_dim* is small enough to be broadcast (after it has been filtered) and *store_returns* has a secondary index on its join predicate. Finally, pilot-run and cost-based will miss the opportunity for choosing INL since the *store_returns* joined with the dimension table and derives from intermediate data; thus the needed secondary index does not exist anymore. Consequently, the difference in the performance against the dynamic approach is even bigger.

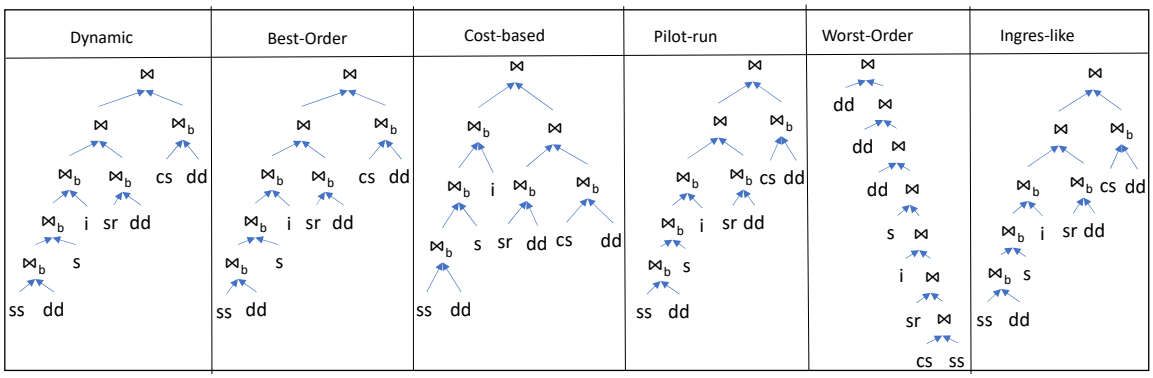
TPC-H

Query 9: Dynamic optimization leads to the choice of INL for the join between *lineitem* and *part*. Thus, the query executes much faster than in the previous section. The same happens with all other approaches apart from the pilot-run in which, similarly to the previous query, *lineitem* does not have a secondary index anymore, thus leading to a performance degradation compared to the dynamic approach.

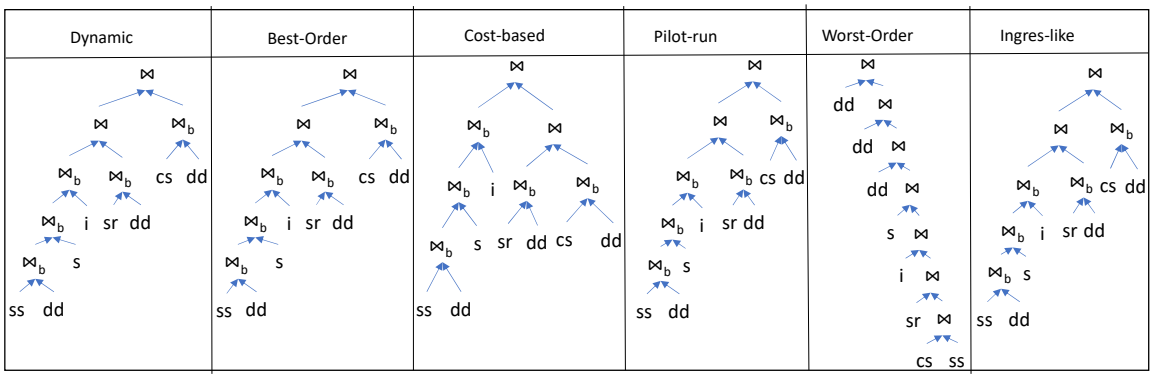
Query 8: This is a case where the INL cannot be triggered for any of the approaches. For example, in the cost-based approach, when *lineitem* and *part* are joined, although there is a secondary index on the *lineitem* predicate and *part* is filtered, the latter is not small enough to be broadcast. In the other approaches, in *supplier* $\bowtie$ *nation* the nation does not have a filter on it; hence, although all the other requirements are met, a simple broadcast will be better because scanning the whole dataset once is preferred to performing too many index lookups.

3.5.4 Discussion

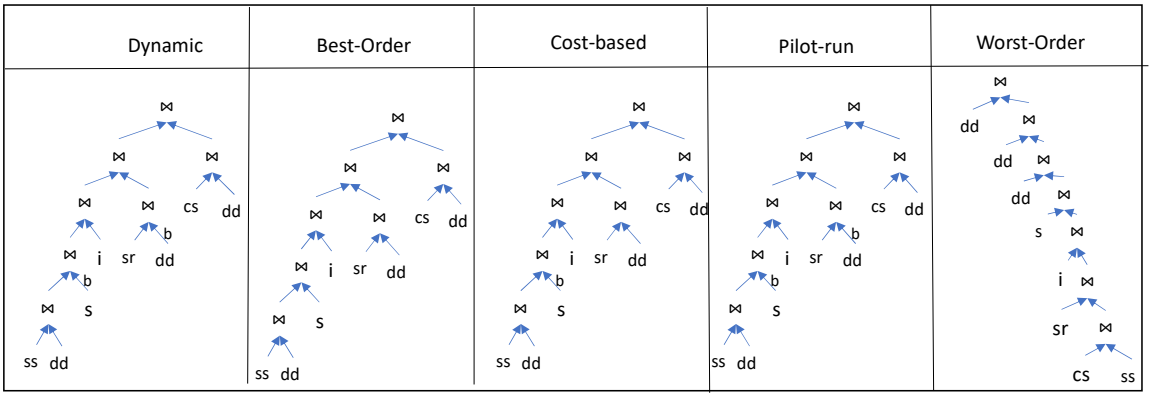
The results of our evaluation showcase the superiority of our dynamic optimization approach against traditional optimization and state-of-the-art techniques. Figure 3.15 shows the spread and centers of all query optimization approaches compared to the dynamic approach. Measures of spread include interquartile range and mean and centers the mean



(a) Scale Factor 10

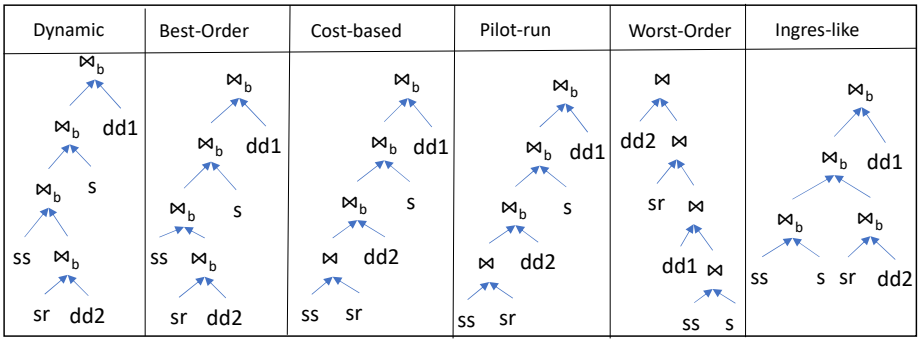


(b) Scale Factor 100

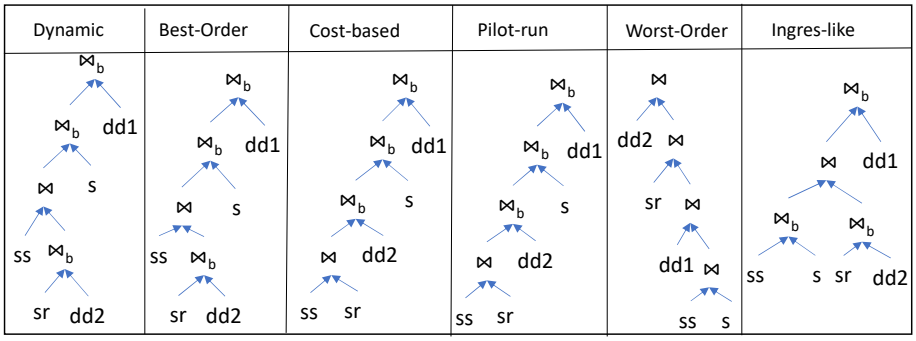


(c) Scale Factor 1000

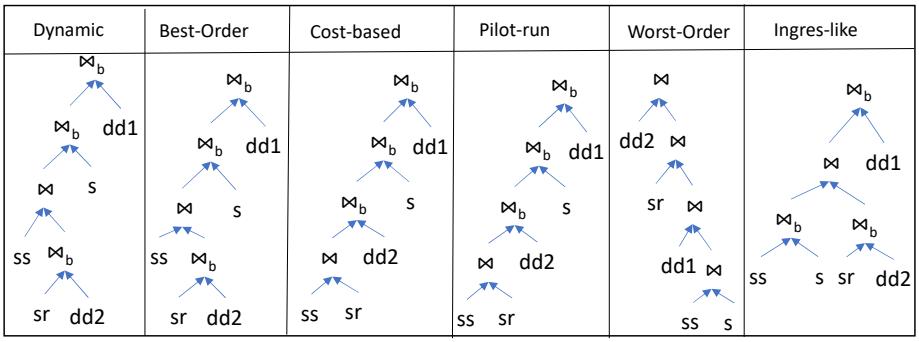
Figure 3.8: Plans Generated for Query 17, Figure 3.6



(a) Scale Factor 10

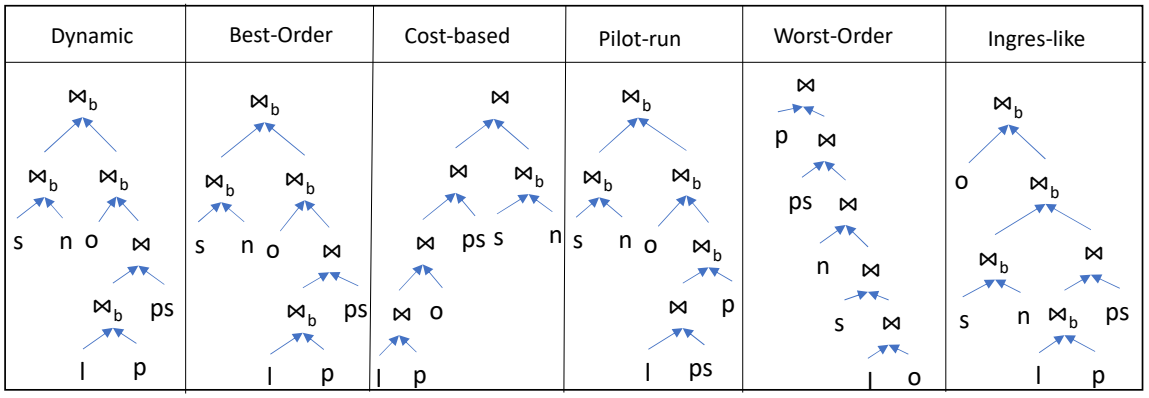


(b) Scale Factor 100

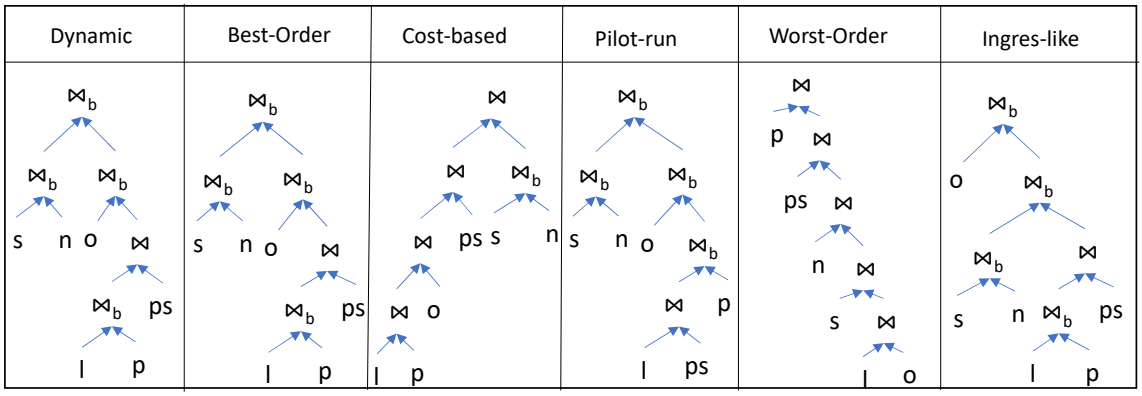


(c) Scale Factor 1000

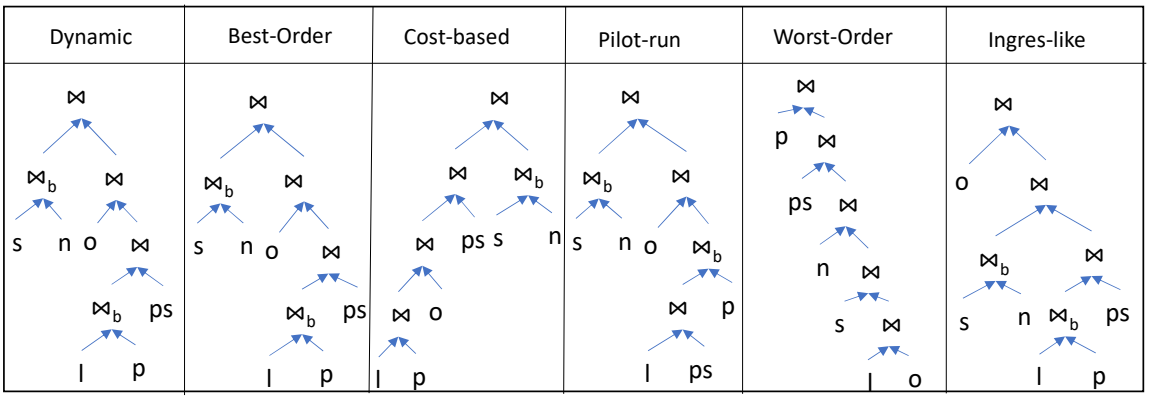
Figure 3.9: Plans Generated for Query 50, Figure 3.6



(a) Scale Factor 10

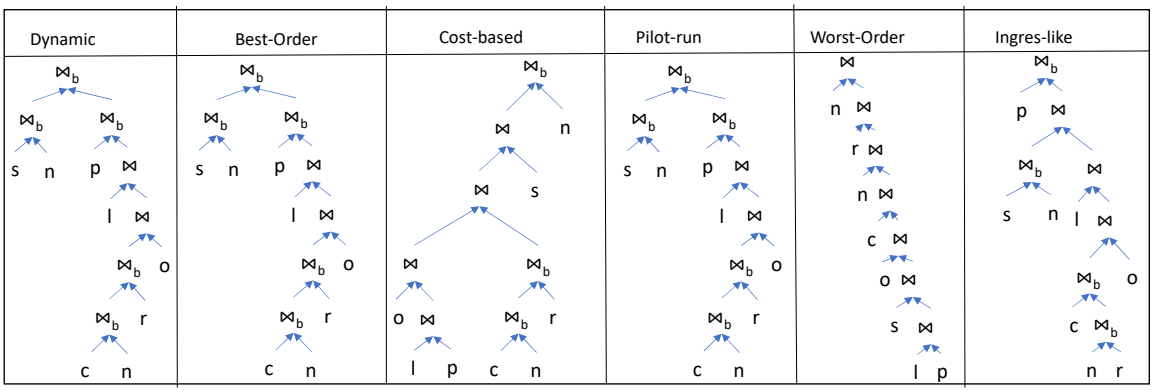


(b) Scale Factor 100

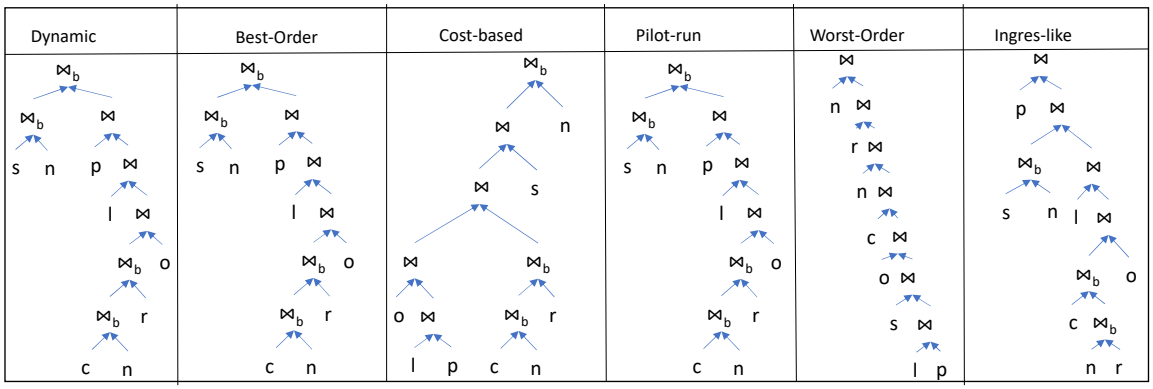


(c) Scale Factor 1000

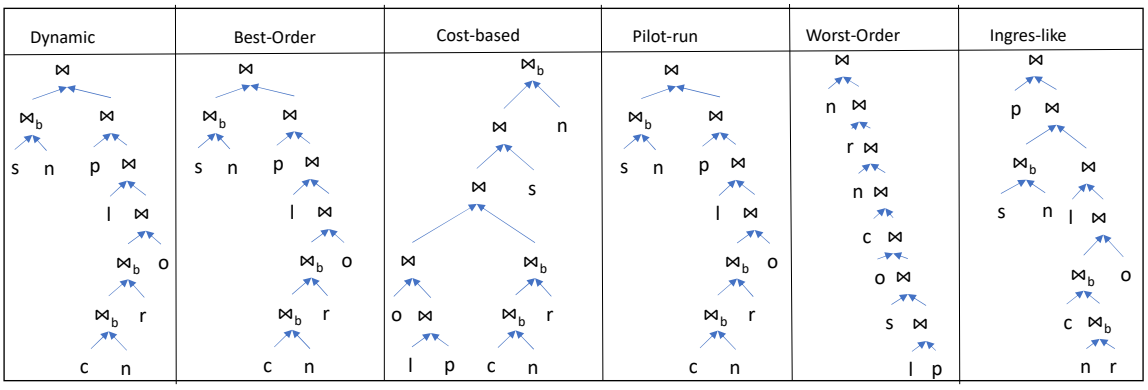
Figure 3.10: Plans Generated for Query 9, Figure 3.6



(a) Scale Factor 10

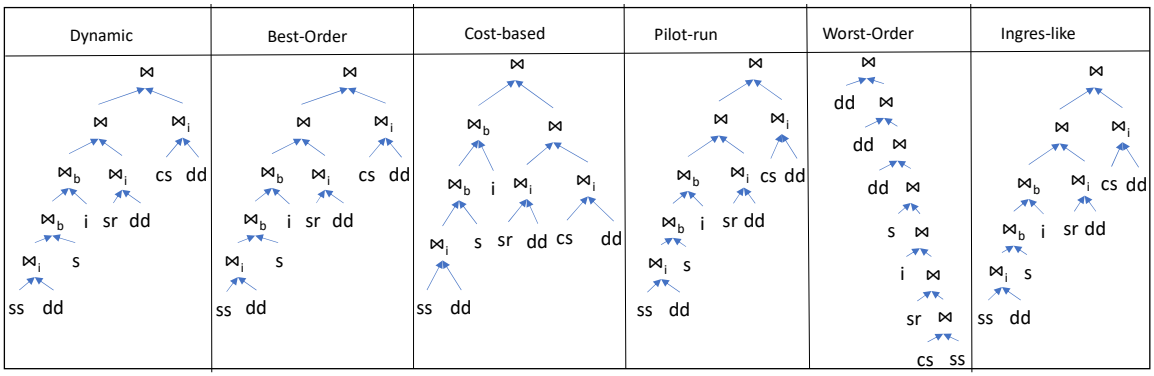


(b) Scale Factor 100

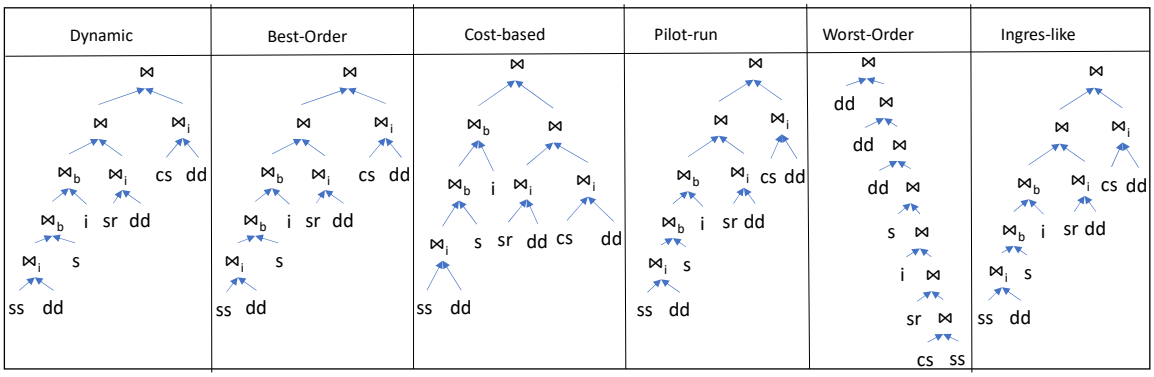


(c) Scale Factor 1000

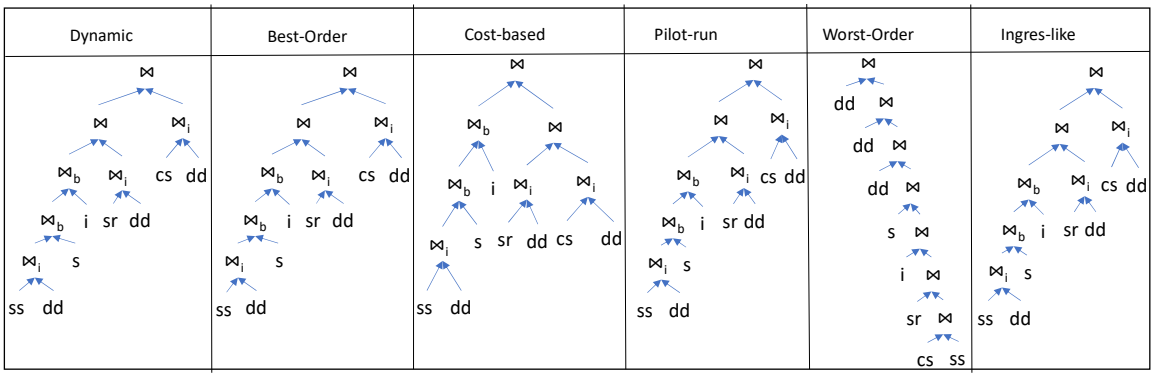
Figure 3.11: Plans Generated for Query 8, Figure 3.6



(a) Scale Factor 10

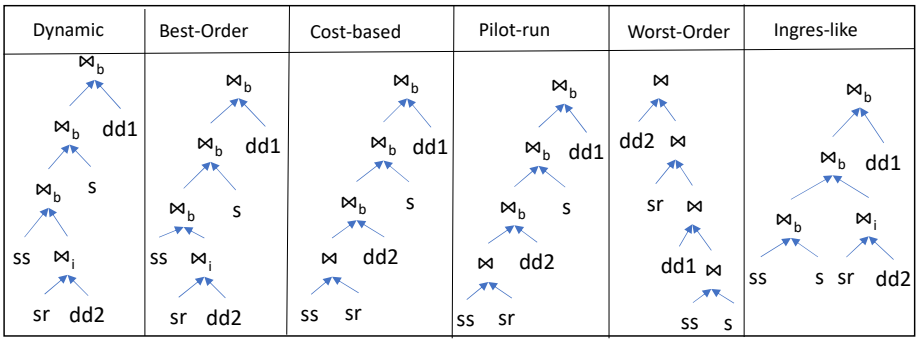


(b) Scale Factor 100

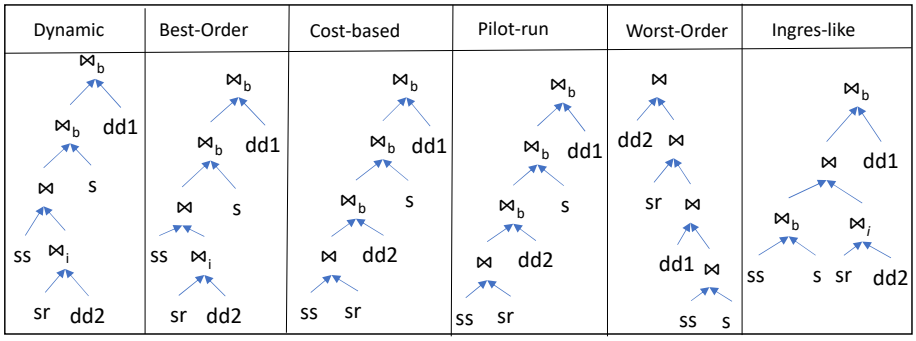


(c) Scale Factor 1000

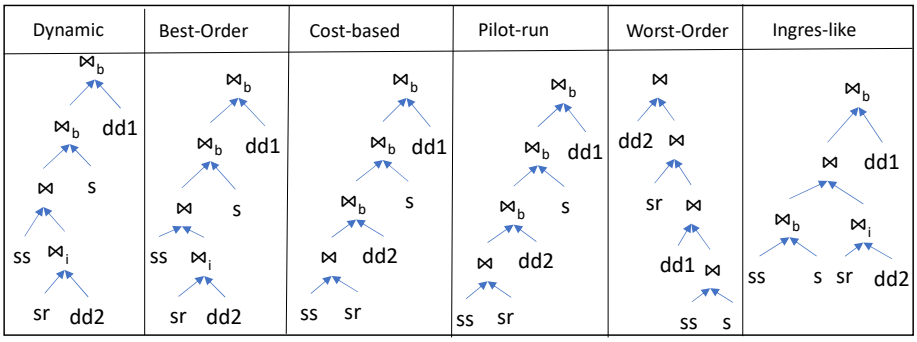
Figure 3.12: Plans Generated for Query 17, Figure 3.7



(a) Scale Factor 10

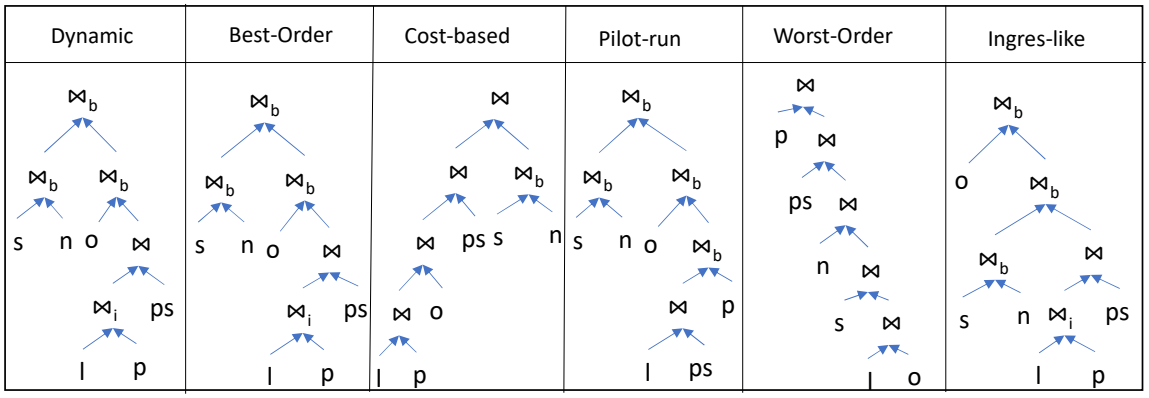


(b) Scale Factor 100

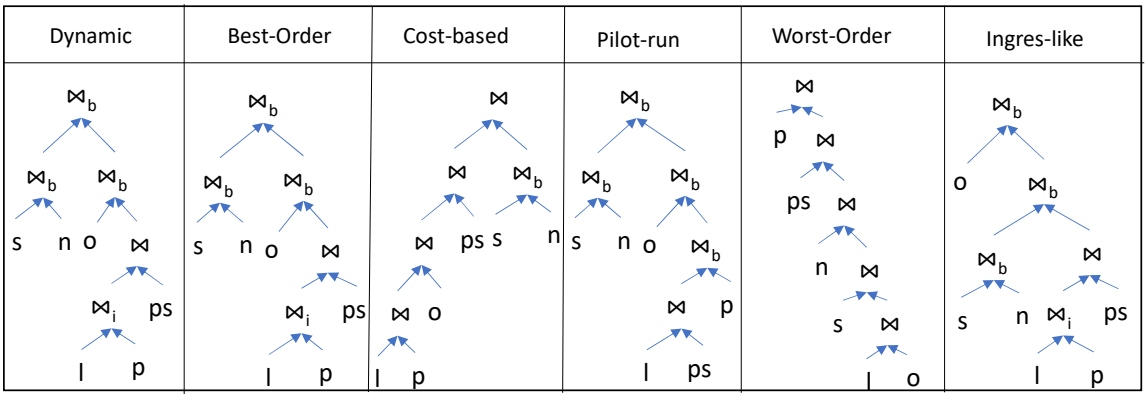


(c) Scale Factor 1000

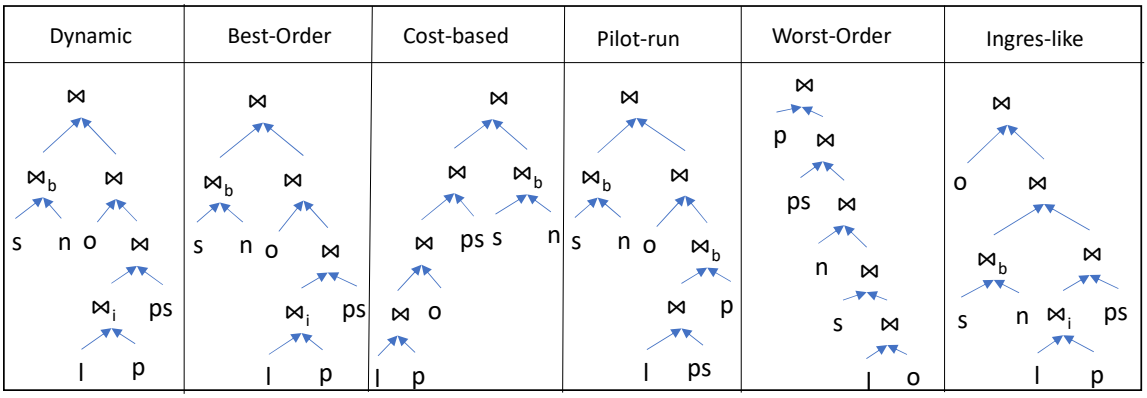
Figure 3.13: Plans Generated for Query 50, Figure 3.7



(a) Scale Factor 10



(b) Scale Factor 100



(c) Scale Factor 1000

Figure 3.14: Plans Generated for Query 9, Figure 3.7

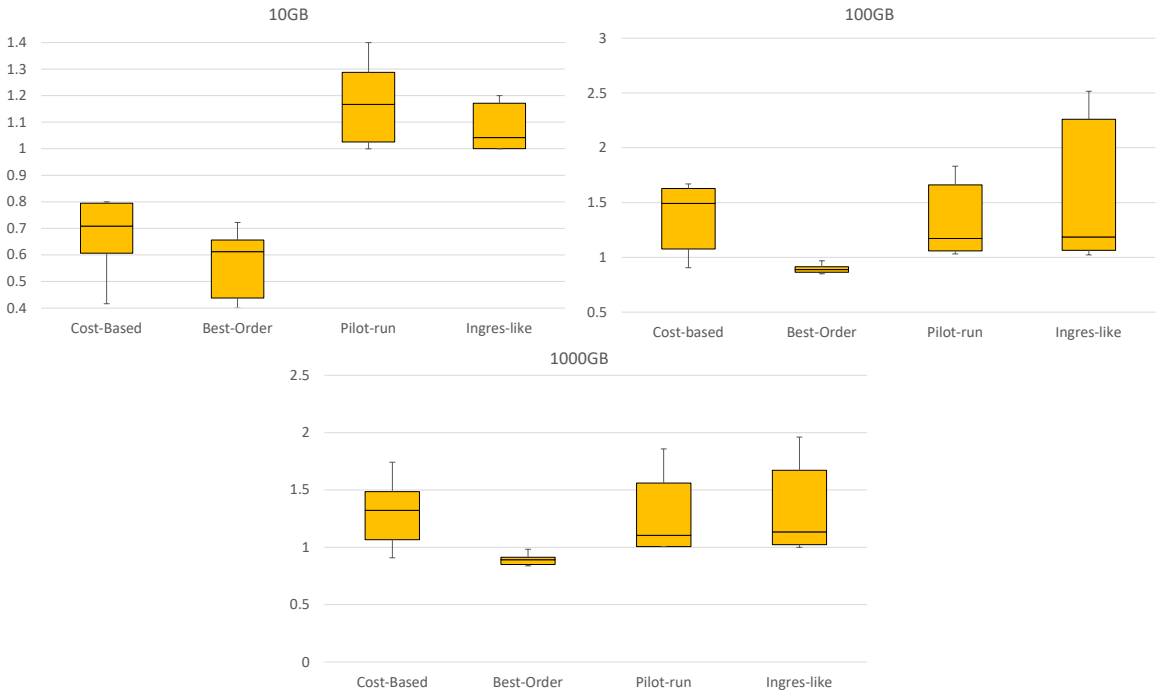


Figure 3.15: Metrics for spread and centers of all query optimization approaches compared to the dynamic approach.

and median. It is worth mentioning that the best improvement is observed for the 100GB dataset size. When the base dataset is large enough, a wrong execution plan chosen by traditional optimizers is noticeable and at the same time the broadcast join has a higher possibility of being picked by our approach due to accurate selectivity estimations (post execution of predicates). For the 1000GB dataset size, we observed less improvement with our approach (see figure 3.15), as broadcast joins are limited, and the intermediate results are larger leading to a larger I/O cost. Nevertheless, we were still better than all the other approaches. For the 10GB size, we have the least improvement (there are even cases where we are worse than cost-based) because the base datasets are very small in size and the overhead imposed by the intermediate data materialization is noticeable. A further interesting observation is that most of the optimal plans are bushy joins, meaning that even if both

inputs have to be constructed before the join is performed, forming the smaller intermediate join results brings more benefits to the query execution.

With respect to the overhead derived by our dynamic optimization techniques, we note that although in the worst case (scale factor 1000) the cost can be expensive, in most cases our plans are still faster than the plans produced by traditional optimizers.

3.6 Conclusions

In this chapter we have investigated the benefits of using dynamic query optimization in big data management systems. We described how we decompose a submitted query into several subqueries with the ultimate goal of integrating re-optimization points to gather statistics on intermediate data and refine the plan for the remaining query. Although our work concentrates on complex join queries, we also treat multiple selective predicates and predicates with parameterized values and UDFs, as part of the re-optimization process. That way, in addition to the benefit of gathering information about the cardinality of intermediate data, we also get more accurate estimations about the sizes of filtered base datasets. We chose AsterixDB to implement our techniques as it is a scalable BDMS optimized to execute joins in a pipeline. We were able to showcase that, even though it blocks the pipelining feature and introduces intermediate results, our approach still gives almost always the best performance.

We evaluated our work by measuring the execution time of different queries and comparing our techniques against traditional static cost-based optimization and the default AsterixDB query execution approach and we proved its superiority. When querying big

data, it pays to get good statistics by allowing re-optimization points since a small error in estimating the size of a big dataset can have much more drastic consequences on query performance than the overhead introduced. Nevertheless, our approach performs at its best when complex predicates are applied to the base datasets of a query or the join conditions are between fact tables (leading to skewness in selectivity and join result estimation accordingly).

In future research we wish to explore ways to address more complex UDFs in our dynamic optimization approach. Further, we want to exploit the benefits of dynamic optimization when other operators (i.e group-by, order by, etc.) are included in the query. Although more re-optimization points make our technique more accurate and robust, they also increase its overhead. Consequently, it would be interesting to explore (through a cost model) the trade-off of facilitating the dynamic optimization approach but with fewer re-optimizations and still obtain accurate results. Finally, runtime dynamic optimization can also be used as a way to achieve fault-tolerance by integrating checkpoints. That would help the system to recover from a failure by not having to start over from the beginning of a long-running query.

Chapter 4

A Parallel and Scalable Processor for JSON Data

4.1 Introduction

The Internet of Things (IoT) has enabled physical devices, buildings, vehicles, smart phones and other items to communicate and exchange information in an unprecedented way. Sophisticated data interchange formats have made this possible by leveraging their simple designs to enable low overhead communication between different platforms. Initially developed to support efficient data exchange for web-based services, JSON has become one of the most widely used formats evolving beyond its original specification. It has emerged as an alternative to the XML format due to its simplicity and better performance [66]. It has been used frequently for data gathering [56], motion monitoring [42], and in data mining applications [60].

When it comes time to query a large repository of JSON data, it is imperative to have a scalable system to access and process the data in parallel. In the past there has been some work on building JSONiq add-on processors to enhance relational database systems, e.g. Zorba [6]. However, those systems are optimized for single-node processing.

More recently, parallel approaches to support JSON data have appeared in systems like MongoDB [14] and Spark [11]. Nevertheless, these systems prefer to first load the JSON data and transform them to their internal data model formats. On the other hand systems like Sinew [71] and Dremel [64] cannot query raw JSON data. They need a pre-processing phase to convert the input file into a readable binary for them (typically Parquet [7]). They can then load the data, transform it to their internal data model and proceed with its further processing. The above efforts are examples of systems that can process JSON data by converting it to their data format, either automatically, during the loading phase, or manually, following the pre-processing phase. In contrast, our JSONiq processor can immediately process its JSON input data without any loading or pre-processing phases. Loading large data files is a significant burden for the overall system's execution time as our results will show in the experimental section. Although, for some data, the loading phase takes place only in the beginning of the whole processing, in most real-time applications, it can be a repetitive action; data files to be queried may not always been known in advance or they may be updated continuously.

Instead of building a JSONiq parallel query processor from scratch, given the similarities between JSON and XQuery, we decided to take advantage of Apache VXQuery [35, 9], an existing processor that was built for parallel and scalable XQuery processing.

We chose to support the JSONiq extension to XQuery language [12] to provide the ability to process JSON data. XQuery and JSONiq have certain syntax conflicts that need to be resolved for a processor to support both of them, so we enhanced VXQuery with the *JSONiq extension* to the XQuery language, an alteration of the initial JSONiq language designed to resolve the aforementioned conflicts [13].

In extending Apache VXQuery, we introduce three categories of JSONiq rewrite rules (*path expression*, *pipelining*, and *group-by* rules) to enable parallelism via pipelining and to minimize the required memory footprint. A useful by-product of this work is that the proposed group-by rules turn out to apply to both XML and JSON data querying. Through experimentation, we show that the VXQuery processor augmented with our JSoniq rewrite rules can indeed query JSON data without adding the overhead of the loading phase used by most of the state-of-the art systems.

The rest of this chapter is organized as follows: Section 4.2 presents the existing work on JSON query processing, while Section 4.3 introduces the specific optimizations applied to JSON queries and how they have been integrated into the current version of VXQuery. The experimental evaluation appears in Section 4.4. Section 4.5 summarizes the techniques described in this chapter and presents directions for future research.

4.2 Related Work

Previous work on querying data interchange formats has primarily focused on XML data [63]. Nevertheless there has been considerable work for querying JSON data. One of the most popular JSONiq processors is Zorba [6]. This system is basically a virtual machine

for query processing. It processes both XML and JSON data by using the XQuery and JSONiq languages respectively. However, it is not optimized to scale onto multiple nodes with multiple data files, which is the focus of our work. In contrast, Apache VXQuery is a system that can be deployed on a multi-node cluster to exploit parallelism.

A few parallel approaches for JSON data querying have emerged as well. These systems can be divided into two categories. The first category includes SQL-like systems such as Jaql [30], Trill [36], Drill [10], Postgres-XL [15], MongoDB [14] and Spark [25], which can process raw JSON data. Specifically, they have been integrated with well-known JSON parsers like Jackson [5]. While the parser reads raw JSON data, it converts it to an internal (table-like) data model. Once the JSON file is in a tabular format, it can then been processed by queries. Our system can also read raw JSON data, but it has the advantage that it does not require data conversion to another format since it directly supports JSON's data model. Queries can thus be processed on the fly as the JSON file is read. It is also worthwhile mentioning that Postgres-XL (a scalable extension to PostgreSQL [16]) has a limitation on how it exploits its parallelism feature. Specifically, while it scales on multiple nodes it is not designed to scale on multiple cores. On the other hand, our system can be multinode and multicore at the same time. In the experimental section we show how our system compares with two representatives from this category (MongoDB and Spark).

We note that AsterixDB [17], can process JSON data in two ways. It can either first load the file internally (like the systems above) or, it can access the file as external data without the need of loading it. However, in both cases and in contrast to our system,

AsterixDB needs to convert the data to its internal ADM data model. In our experiments we compare VXQuery with both variations of AsterixDB.

Systems in the second category (e.g. Sinew [71], Argo [37] and Oracle's system [61]) cannot process raw JSON data and thus need an additional pre-processing phase (hence an extra overhead than the systems above). During that phase, a JSON file is converted to a binary or Parquet ([7]) file that is then fed to the system for further transformation to its internal data model before query processing can start.

Systems like Spark and Argo process their data in-memory. Thus, their input data sizes are limited by a machine's memory size. Recently, [58] presents an approach that pushes the filters of a given query down into the JSON parser (Mison). Using data-parallel algorithms, like SIMD vectorization and Bitwise Parallelism, along with speculation, data not relevant to the actual query is filtered out early. This approach has been added into Spark and improves its JSON performance. Our work also prunes irrelevant data, but does so by applying rewrite rules. Since the Mison code is not available yet, we could not compare with them in detail; we also need to note that Mison is just a parallel JSON parser for JSON data. In contrast, VXQuery is an integrated processor that can handle the querying of both JSON and XML data (regardless of how complex the query is).

As opposed to the aforementioned systems, our work builds a new JSONiq processor that leverages the architecture of an existing query engine and achieves high parallelism and scalability via the employment of rewrite rules.

4.3 JSON Query Optimization

The JSONiq rewrite rules are divided into three categories: the Path Expression, Pipelining, and Group-by Rules. The first category removes some unused expressions and operators, as well as streamlining the remaining path expressions. The second category reduces the memory needs of the pipeline. The last category

focuses on the management of aggregation, which also contains the group-by feature (added to VXQuery in the XQuery 3.0 specification). For all our examples, we will consider the bookstore structure example depicted in Listing 4.1.

```
{
  "bookstore": {
    "book": [
      {
        "-category": "COOKING",
        "title": "Everyday Italian",
        "author": "Giada De Laurentiis",
        "year": "2005",
        "price": "30.00"
      },
    ...
    ]
 }
```

Listing 4.1: Bookstore JSON File

4.3.1 Path Expression Rules

The goal of the first category of rules is to enable the unnesting property. This means that instead of creating a sequence of all the targeted items and processing the whole sequence, we want to process each item separately as it is found. This rule opens up opportunities for pipelining since each item is passed to the next stage of processing as the previous step is completed.

```
json-doc("books.json")("bookstore")("book")()
```

Listing 4.2: Bookstore query

The example query in Listing 4.2 asks for all the books appearing in the given file. Specifically, it reads data from the JSON file ("book.json") and then, the *value* expression is applied twice, once for the bookstore object (("bookstore")) and once for the book object (("book")). In this way, it is ensured that only the matching objects of the file will be stored in memory. The value of the book object is an array, so the *keys-or-members* expression (()) applied to it returns all of its items. To process this expression, we first store in a tuple all of the objects from the array and then we iterate over each one of them.

The result that is distributed at the end is each book object separately.

```
DISTRIBUTE-RESULT( $$9 )
UNNEST( $$9:iterate($$8) )
ASSIGN( $$8:(keys-or-members($$2)) )
ASSIGN( $$2:value(value(json-doc(promote(data("books.json"),
string)),"bookstore"),"book"))
EMPTY-TUPLE-SOURCE
```

Figure 4.1: Original Query Plan

In more detail, we can describe the aforementioned process in terms of a logical query plan that is returned from VXQuery (Figure 4.1). It follows a bottom-up flow, so the first operator in the query plan is the EMPTY-TUPLE-SOURCE leaf operator. The empty tuple is extended by the following ASSIGN operator, which consists of a *promote* and a *data* expression to ensure that the json-doc argument is a string. Also, the two *value* expressions inside it verify that only the book array will be stored in the tuple.

The next two operators depict the two steps of the processing of the *keys-or-members* expression. The first operator is an ASSIGN, which evaluates the expression to extend its input tuple. Since this expression is applied to an array, the returned tuple includes all of the objects inside the array. Then, the UNNEST operator applies an iterate expression to the tuple and returns a stream of tuples including each object from the array. s.

The final step according to the query plan is the distribution of each object returned from the UNNEST. From the analysis above, we can observe that there are opportunities to make the logical plan more efficient. Specifically, we observe that there is no need for two processing steps of *keys-or-members*.

Originally, the tuple with all the book objects produced by the keys-or-members expression flows into the UNNEST operator whose iterate expression will return each object in a separate tuple. Instead, we can merge the UNNEST with the keys-or-members expression. That way, each book object is returned immediately when it is found.

Finally, to further clean up our query plan, we can remove the *promote* and *data* expressions included in the first ASSIGN operator. The fully optimized logical plan is depicted in Figure 4.2.

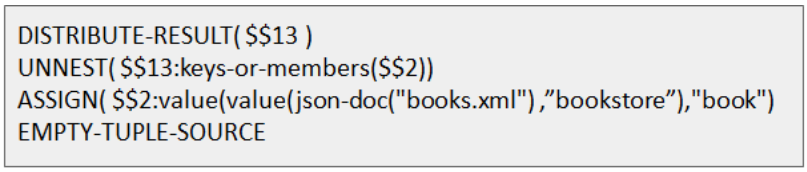


Figure 4.2: Updated Query Plan

The new and more efficient plan opens up opportunities for pipelining since when a matching book object is found, it is immediately returned and, at the same time, passed to the next stage of the process.

4.3.2 Pipelining Rules

This type of rule builds on top of the previous rule set and considers the use of the DATASCAN operator along with the way to access partitioned-parallel data. The sample query that we use is depicted in Listing 4.3.

```
collection("/books")("bookstore")("book")()
```

Listing 4.3: Bookstore Collection Query

According to the query plan in Figure 4.3, the ASSIGN operator takes as input data source a collection of JSON files, through the *collection* expression. Then, UNNEST *iterate* iterates over the collection and outputs each single file. The two value expressions integrated into the second ASSIGN output a tuple source filled with all the book objects

produced by the whole collection. The last step of the query plan (created in the previous section) is implemented by the keys-or-members expression of the UNNEST operator, which outputs each single object separately.

```
DISTRIBUTE-RESULT( $$13 )
UNNEST( $$13:keys-or-members($$6))
ASSIGN( $$6:value(value($$4,"bookstore"),"book"))
UNNEST( $$4:iterate($$2))
ASSIGN( collection("/books"), $$2)
EMPTY-TUPLE-SOURCE
```

Figure 4.3: Query Plan for a Collection

The input tuple source generated by the *collection* expression corresponds to all the files inside the collection. This does not help the execution time of the query, since the result tuple can be huge. Fortunately, Algebricks offers its DATASCAN operator, which is able to iterate over the collection and forwards to the next operator each file separately. To accomplish this procedure, DATASCAN replaces both the ASSIGN *collection* and the UNNEST *iterate*.

```
DISTRIBUTE-RESULT( $$13 )
UNNEST( $$13:keys-or-members($$4))
ASSIGN( $$4:value(value ( $$2,"bookstore"),"book"))
DATASCAN( collection("/books"), $$2)
EMPTY-TUPLE-SOURCE
```

Figure 4.4: Introduction of DATASCAN

By enabling Algebrick's DATASCAN, apart from pipeline improvement, we also achieve partitioned parallelism. In Apache VXQuery, data is partitioned among the cluster nodes. Each node has a unique set of JSON files stored under the same directory specified

in the *collection* expression. The Algebricks' physical plan optimizer uses these partitioned data property details to distribute the query execution. Adding these properties allows Apache VXQuery to achieve partitioned-parallel execution without any user-level parallel programming.

To further improve pipelining, we can produce even smaller tuples. Specifically,

we extend the DATASCAN operator with a second argument (here it is the book array). This argument defines the tuple that will be forwarded to the next operator.

In the updated query plan (Figure 4.4), the newly inserted DATASCAN is followed by an ASSIGN operator. Inside it, the two *value* expressions populate the tuple source with all the book objects of the file fetched from DATASCAN. We can merge the value expressions with DATASCAN by adding a second argument to it. As a result, the output tuple, which was previously filled with each file, is now set to only have its book objects (Figure 4.5).

```
DISTRIBUTE-RESULT( $$13 )
UNNEST( $$13:keys-or-members($$4))
DATASCAN( collection("/books"), $$4, ("bookstore")("book") )
EMPTY-TUPLE-SOURCE
```

Figure 4.5: Merge *value* with DATASCAN Operator

At this point, we note that by building on the previous rule set, both the query's efficiency and the memory footprint can be further improved. In the query plan in Figure 4.5, DATASCAN *collection* is followed by an UNNEST whose *keys-or-members* expression outputs a single tuple for each book object of the input sequence.

This sequence of operators gives us the ability to merge DATASCAN with *keys-or-members* by extending its second argument. Figure 4.6 shows this action, whose result

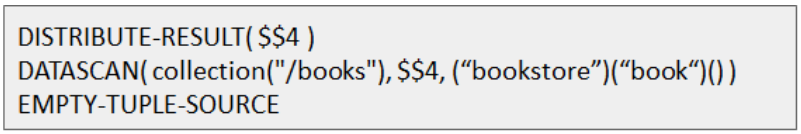


Figure 4.6: Merge keys-or-members with Datascan Operator

is the fetching of even smaller tuples to the next stage of processing. Specifically, instead of storing in DATASCAN's output tuple a sequence of all the book objects of each file in the collection, we store only one object at a time. Thus, query's execution time is improved and Hyracks' dataflow frame size restriction is satisfied.

```
for $x in collection("/books")("bookstore")
("book")()
group by $author:=$x("author")
return count($x("title"))
```

Listing 4.4: Bookstore Count Query

```
for $x in collection("/books")("bookstore")
("book")()
group by $author:=$x("author")
return count(for $j in $x return $j("title"))
```

Listing 4.5: Bookstore Count Query (2nd form)

4.3.3 Group-by Rules

The last category of rules can be applied to both XML and JSON queries since the group-by feature is part of both syntaxes. Group-by can activate rules enabling parallelism in aggregation queries.

The example query that we will use to show how our rules affect aggregation queries is in Listings 4.4 and 4.5. Both forms of the query read data from a collection with book files, group them by author, and then return the number of books written by each author.

```
DISTRIBUTE-RESULT( $$20 )
UNNEST($$20:iterate($$19))
ASSIGN($$19:count(value($$18, “title”)))
ASSIGN($$18:treat(item,$$16))
GROUP-BY($$17:0->$$14){
   AGGREGATE($$16:create_sequence($$13))
}
ASSIGN($$14:data(value($$13, “author”)))
DATASCAN( collection("/books"), $$13, (“bookstore”)(“book“)( ))
EMPTY-TUPLE-SOURCE
```

Figure 4.7: Query Plan with Count Function

In Figure 4.7, the DATASCAN *collection* passes a tuple, for one book object at a time, to ASSIGN. The latter applies the *value* expression to acquire the author's name for the specific object. GROUP-BY accepts the tuple with the author's name (group-by key) and then its inner focus is applied (AGGREGATE) so that all the objects whose author field have the same value will be put in the same sequence.

At this point, ASSIGN *treat* appears; this ensures that the input expression has the designated type. So, our rule searches for the type returned from the sequence created from the AGGREGATE operator. If it is of type item which is the treat type argument, the whole *treat* expression can be safely removed.

As a result, the whole ASSIGN can now be removed since it is a redundant operator (Figure 4.8).

```
DISTRIBUTE-RESULT( $$20 )
UNNEST($$20:iterate($$19))
ASSIGN($$19:count(value($$16, “title”)))
GROUP-BY($$17:0->$$14){
   AGGREGATE($$16:create_sequence($$13))
}
ASSIGN($$14:data(value($$13, “author”)))
DATASCAN( collection("/books"), $$13, (“bookstore”)(“book“)( ))
EMPTY-TUPLE-SOURCE
```

Figure 4.8: Query Plan without *treat* Expression

After the former removal, GROUP-BY is followed by an ASSIGN *count* which calculates the number of book titles (*value* expression) generated by AGGREGATE *sequence*. According to the JSONiq extension to XQuery, *value* can be applied only on a JSON object or array. However, in our case the query plan applies *value* to a sequence, since GROUP-BY aggregates all the records having the same group-by key in a sequence. Thus, ("title") is applied on a sequence.

To overcome this conflict, we convert the ASSIGN to a SUBPLAN operator (Figure 4.9). SUBPLAN's inner focus introduces an UNNEST *iterate* which iterates over AGGREGATE *sequence* and produces a single tuple for each item in the sequence. The inner focus of SUBPLAN finishes with an AGGREGATE along with a count function which incrementally calculates the number of tuples that UNNEST feeds it with.

This conversion not only helps resolving the aforementioned conflict but it also converts the scalar count function to an aggregate one. This means that instead of calculating count on a whole sequence, we can incrementally calculate it as each item of the sequence is fetched.

```
DISTRIBUTE-RESULT( $$20 )
UNNEST($$20:iterate($$19))
SUBPLAN{
   AGGREGATE($$19:count(value($$18, “title”)))
   UNNEST($$18:iterate($$16))
}
GROUP-BY($$17:0->$$14){
   AGGREGATE($$16:create_sequence($$13))
}
ASSIGN($$14:data(value($$13, “author”)))
DATASCAN( collection("/books"), $$13, (“bookstore”)(“book“)( ))
EMPTY-TUPLE-SOURCE
```

Figure 4.9: Convert scalar to aggregation expression

In Figure 4.9, GROUP-BY still creates a sequence in its inner focus, which is the input to SUBPLAN UNNEST. Instead, we can push the AGGREGATE operator of the SUBPLAN down to the GROUP-BY operator by replacing it (Figure 4.10). That way, we exploit the benefits of the aforementioned conversion and have the count function computed at the same time that each group is formed (without creating any sequences). Thus, the new plan is not only smaller (more efficient) but also keeps the pipeline granularity introduced in both of the previous rule sets.

At this point, it is interesting to look at the second format of the query in Listing 4.5. The for loop inside the count function conveniently forms a SUBPLAN operator right above the GROUP-BY in the original logical plan. This is exactly the query plan described in Figure 4.9, which means that in this case we can immediately push the AGGREGATE down to GROUP-BY, without any further transformations.

```
DISTRIBUTE-RESULT( $$20 )
UNNEST($$20:iterate($$16))
GROUP-BY($$17:0->$$14){
   AGGREGATE($$16:count(value($$13,"title")))
}
ASSIGN($$14:data(value($$13, "author")))
DATASCAN( collection("/books"), $$13, ("bookstore")("book")( ))
EMPTY-TUPLE-SOURCE
```

Figure 4.10: Updated Query Plan with Count Function

Now that the count function is converted into an aggregate function, there is another rule introduced in [35] that can be activated to further improve the overall query performance. This rule supports Algebricks' two-step aggregation scheme, which means that each partition can calculate locally the count function on its data. Then, a central node can compute the final result using the data gathered from each partition. Thus, partitioned computation is enabled, which improves parallelism effectiveness.

The final query plan, produced after the application of all the former rules, calculates the count function at the same time that each grouping sequence is built as opposed to first building it and then processing the aggregation function.

4.4 Experimental Evaluation

We have tested our rewrite rules by integrating them into the latest version of Apache VXQuery [9]. Each node has two dual-core AMD Opteron(tm) processors, 8GB of memory, and two 1TB hard drives. For the multi-node experiments we built a cluster of up to nine nodes of identical configuration. We used a real dataset with sensor data and a variety of queries, described below in Sections 4.4.1 and 4.4.2 respectively. We repeated each query

five times. The reported query time is an average of the five runs. We first consider single-node experiments and include measurements for execution time (before and after applying our rewrite rules) and for speed-up. For the multi-node experiments we measure response time and scale-up over different numbers of nodes. We, also, include comparisons with Spark SQL and MongoDB that show that the overhead of their loading phase is non-negligible. Finally, we compare with AsterixDB which has the same infrastructure as our system; in particular we compare with two approaches, one that sees the input as an external dataset (depicted in the figures as AsterixDB) and one that first loads the file internally (depicted as AsterixDB(load)).

4.4.1 Dataset

The data used in our experiments are publicly available from the National Oceanic and Atmospheric Administration (NOAA) [55]. The Daily Global Historical Climatology Network (GHCN-Daily) dataset was originally in dly format and was converted to its equivalent NOAA web service JSON representation.

```
{
  "root": [
    {
      "metadata": {
        "count":29,
      },
      "results": [
        {
          "date":"20142512T00:00",
          "dataType":"WIND",
          "station":"GSW957859",
          "value":30
        },
        ...
      ]
    },
    ...
  ]
}
```

Listing 4.6: Example JSON Sensor File Structure

Listing 4.6 shows an example JSON sensor file structure. All records are wrapped into a JSON item, specifically array, called `"root"`. Each member of the `"root"` array is an object item which contains the object `"metadata"` and the array `"results"`. The latter stores various measurements organized in individual objects. A specific measurement includes the date, the data type (a description of the measurement, with typical values

being TMIN, TMAX, WIND etc.), the station id where the measurement was taken, and the actual measurement value. The `"count"` object included into the `"metadata"` denotes the number of measurements objects in the accompanying `"results"` array. Typically a `"results"` array contains measurements from a given station for one month (i.e. typically 30 measurements). A sensor file contains only one `"root"` array which may consist of several `"results"` (measurements from the same or different stations) accompanied by their `"metadata"`.

Sensor file sizes vary from 10MB to 2GB. Each node holds a collection of sensor files; the size of the collection varies from 100MB to 803GB. The collection size is varied throughout the experiments and is cited explicitly for each experiment.

In our experiments, we assume that the data has already been partitioned among the existing nodes.

4.4.2 Query Types

```
for $r in collection("/sensors")("root")()("results")()
let $datetime := dateTime(data($r("date")))
where year-from-dateTime($datetime) ge 2003
  and month-from-dateTime($datetime) eq 12
  and day-from-dateTime($datetime) eq 25
return $r
```

Listing 4.7: Selection Query (Q0)

We evaluated our newly implemented rewrite rules by evaluating different types of queries including selection (Q0, Q0b), aggregation (Q1, Q1b) and join-aggregation queries (Q2). The query description follows.

```
for $r in collection("/sensors")("root")()("results")()("date")
let $datetime := dateTime(data($r))
where year-from-dateTime($datetime) ge 2003
  and month-from-dateTime($datetime) eq 12
  and day-from-dateTime($datetime) eq 25
return $r
```

Listing 4.8: Selection Query (Q0b)

Q0: In this query (Listing 4.8), the user asks for historical data from all the weather stations by selecting all December 25 weather measurement readings from 2003 on.

```
for $r in collection ("/sensors") ("root")()
("results")()
where $r("dataType") eq "TMIN"
group by $date:= $r("date")
return count($r("station"))
```

Listing 4.9: Aggregation Query (Q1)

Q0b is a variation of Q0 where the input path (1st line in Listing 4.8), is extended by a *value* expression ("date").

Q1: This query (Listing 4.10), finds the number of stations that report the lowest temperature for each date. The grouping key is the date field of each object.

```
for $r in collection ("/sensors") ("root")()
("results")()
where $r("dataType") eq "TMIN"
group by $date:= $r("date")
return count(for $i in $r return $i("station"))
```

Listing 4.10: Aggregation Query (Q1b)

Q1b is a variation of Q1 that has a different returned result structure.

```
avg(
  for $r_min in collection("/sensors")("root")()("results")()
  for $r_max in collection("/sensors")("root")()("results")()
  where $r_min("station") eq $r_max("station")
    and $r_min("date") eq $r_max("date")
    and $r_min("dataType") eq "TMIN"
    and $r_max("dataType") eq "TMAX"
  return $r_max("value") - $r_min("value")
) div 10
```

Listing 4.11: Join-Aggregation Query (Q2)

Q2: This self-join query (Listing 4.11) joins two large collections, one that maintains the daily minimum temperature per station and one that contains the daily maximum temperature per station. The join is on the station id and date and finds the daily temperature difference per station and returns the average difference over all stations.

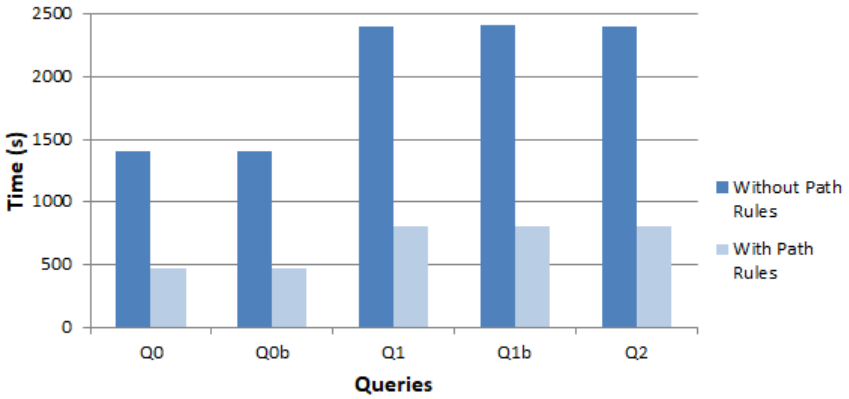


Figure 4.11: Execution Time before and after Path Expression Rules

4.4.3 Single Node Experiments

To explore the effects of the various rewrite rules we first consider a single node, one core environment (i.e. one partition) and progressively enable the different sets of rules.

We start by considering just the *path expression rules*. Figure 4.11 shows the execution time for all five queries with and without these rules. For this experiment, we used a 400MB collection of sensor files (each of size 10MB). Note that for these experiments we used a relatively small collection size since without the JSONiq rules Hyracks would need to process the whole (possibly large) file thus slowing its performance. The application of the Path Expression Rules results to a clear improvement of the execution time for all queries. These rules decrease the buffer size between operators since large sequences of objects are avoided. Instead, each object is passed on to the next operator separately, resulting in faster query execution.

Using the same dataset and having enabled the Path Expression rules, we now examine the effect of adding the *Pipelining* rules (Figure 4.12). We observe that in all cases

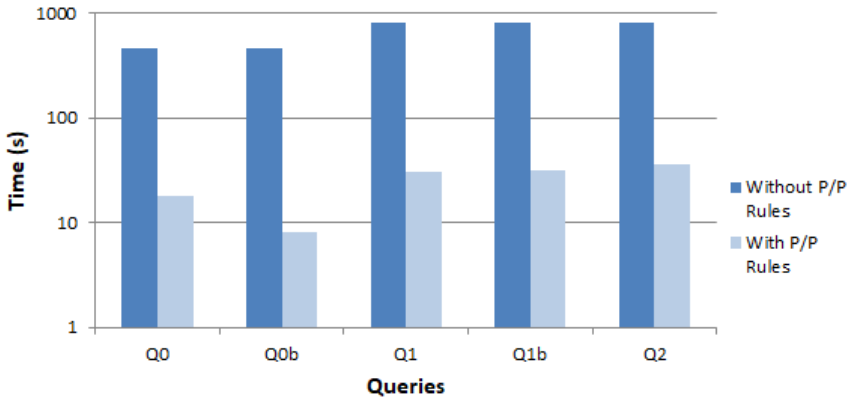


Figure 4.12: Execution Time (logscale) before and after the Pipelining Rules

the pipelining rules provide a drastic improvement (note the logarithmic scale!), speeding queries up by about two orders of magnitude. Applying these rules decreases the initial buffering requirement since we don't store the whole JSON document anymore, but just the matching objects. It is worth noting that the best performance is achieved by Q0b. Q0b stores in memory only the parts of the objects whose key field is "date". By focusing only on the "date", this gives the DATASCAN operator the opportunity to iterate over much smaller tuples than Q0. Clearly, the smaller the argument given to DATASCAN, the better for exploiting pipelining.

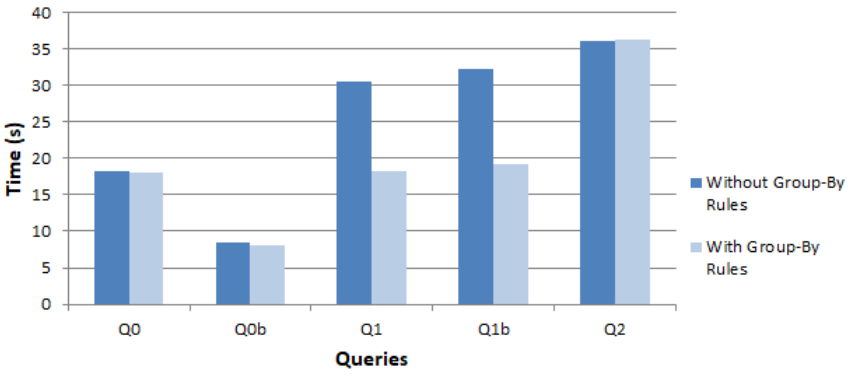


Figure 4.13: Execution Time before and after Group-by Rules

Having enabled both the path expression and the pipelining rules, we proceed considering the effect of adding the *Group-by* rules. The results are depicted in Figure 4.13. Clearly Q0, Q0b and Q2 are not affected since the Group-by rules do not apply. The Group-by rules improve the performance of both queries Q1 and Q1b. The improvement for both queries comes from the same rule, the rule that pushes the COUNT function inside the group-by. We note that the second Group-by rule, the one performing conversion, applies only to Q1; we do not enjoy an improvement from the conversion rule here because Q1b is already written in an optimized way. It is worth mentioning that the efficiency of the group-by rules depends on the cardinality of the groups created by the query. Clearly, the larger the groups, the better the observed improvement.

To study the effectiveness of all of the rewrite rules as the partition size increases, we ran an experiment where we varied the collection size from 100MB to 400MB. Figure 4.14 (again with a log scale) depicts the execution time for query Q1 both before and after applying all three sets of rewrite rules. We chose Q1 here because it is indeed affected by all of the rules. We can see that the system scales proportionally with the dataset size and that the application of the rewrite rules results in a huge query execution time improvement.

From the above experiments, we can conclude that the Pipelining rules provide the most significant impact. It is also worth noting that the execution of the rewrite rules during query compilation adds a minimal overhead (just a few msec) to the overall query execution cost.

Single node Speed-up: To test the system's single node speed up, we used a dataset larger than our node's available memory space (8GB). Specifically, we used 88GB

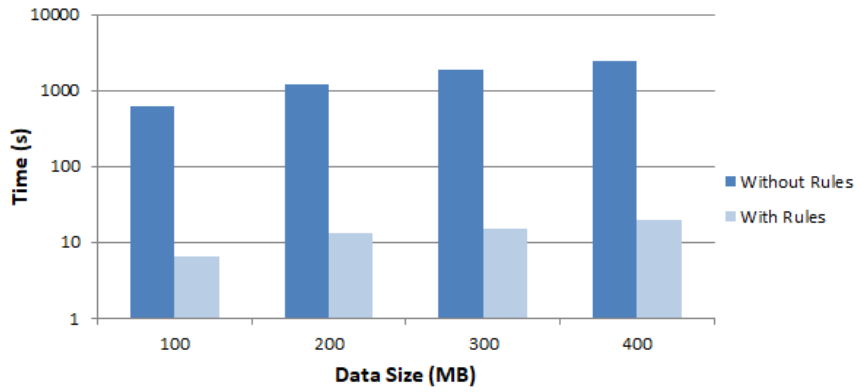


Figure 4.14: Execution Time (logscale) for Q1 before and after Rewrite Rules for different Data Sizes

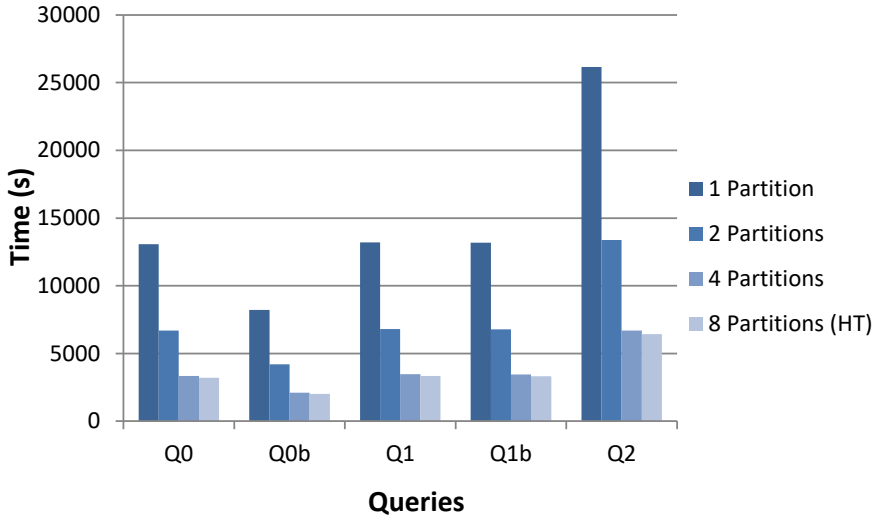


Figure 4.15: Single Node Speed-up

of JSON data, which we progressively divided from one up to eight partitions (our CPU has 4 cores and supports up to 8 hyperthreads). Each partition corresponds to a thread. The results are shown in Figure 4.15.

For up to 4 partitions and for almost all query categories, we achieve good speed-up since our observed execution time is reduced by a factor close to the number of threads that are being used. On the other hand, when using 8 hyperthreaded partitions we observe no

performance improvements and in some cases a slightly worst execution time. This is because our processing is CPU bound (due to the JSON parsing), hence the two hyperthreads are effectively run in sequence (on a single core). In summary, we see the best results when we match the number of partitions to the number of cores.

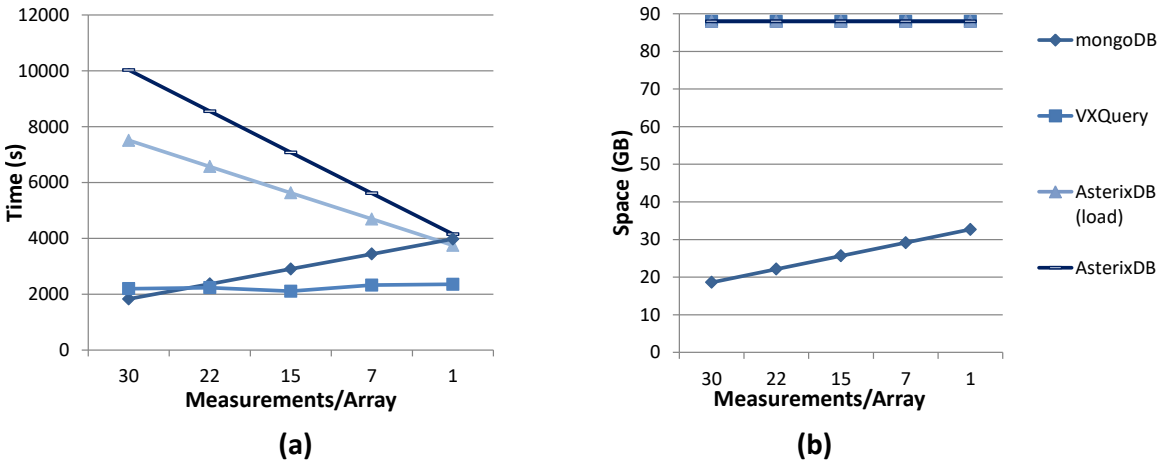


Figure 4.16: (a) Execution Time and (b) Space Consumption for Different Measurement Sizes per Array

Comparison with MongoDB and AsterixDB: When comparing our performance against MongoDB and AsterixDB we observed that the performance of these systems is affected by the structure of the input JSON file. For example, a file with the structure of Listing 4.6 will be perceived by MongoDB and AsterixDB as a single, large document. Since MongoDB and AsterixDB are optimized to work with smaller documents (MongoDB has in addition a document size limitation of 16MB), to make a fair comparison we examined their performance while varying the number of documents per file.

We first unwrapped all the JSON items inside `"root"` (Listing 4.6). This results to many individual documents per file, each document containing a `"metadata"` JSON object and its corresponding `"results"` JSON array (typically with 30 measurements). We

further manipulated the number of documents per file by varying the number of member objects (measurements) inside the `"results"` array from 30 (one month of measurements per document) to 1 (one day/measurement per document).

Figure 4.16.a depicts the query time performance for query Q0b for VXQuery, MongoDB, AsterixDB and AsterixDB(load); the space used by each approach appears in Figure 4.16.b. The total size of the dataset is 88GB and we vary the measurements per JSON array.

In contrast to MongoDB and the AsterixDB approaches, we note that the performance of VXQuery is independent of the number of documents per file. MongoDB has better query time for larger documents (30 measurements per array). Since MongoDB performs compression per document, larger documents allow for better compression and thus query time performance. This can also be seen in figure 4.16.b, where the space requirements increase as the document becomes smaller (and thus less compression is possible). The space for both AsterixDB approaches and VXQuery is independent from the document size (which is to be expected as currently these systems do not use compression).

AsterixDB shows a different query performance behavior than MongoDB. Its best performance is achieved for smaller document sizes (one measurement per document). Since it shares the same infrastructure as VXQuery, the difference in its performance relative to VXQuery is due to the lack of the JSONiq Pipeline Rules. Without them, the system waits to first gather all the measurements in the array before it moves them to the next stage of processing. This holds for both AsterixDB and AsterixDB(load). Among the two approaches, the AsterixDB(load) approach has better query performance since it is

Measurements/Array	30	22	15	7	1
MongoDB	9000	11703	14443	17146	19876
AsterixDB (load)	24659	23987	24205	24547	24612

Table 4.1: Loading Time in sec for AsterixDB (load) and MongoDB for Different Record Sizes

optimized to work better for data that is already in its own data model. Interestingly, for the smaller document sizes (where compression is limited), AsterixDB and MongoDB have similar query performance. For larger document sizes their query performance difference seems to be directly related to their data sizes. For example, with 30 measurements per document, MongoDB uses about 4.5 times less space due to compression and has about 4 times less query time than AsterixDB(load).

Table 4.1 depicts the loading times for MongoDB and AsterixDB(load) for different measurements/array (in contrast there is no loading time for AsterixDB and VXQuery). The different loading times can also be explained by the space consumed by MongoDB and AsterixDB(load) (Figure 4.16.b). Specifically, MongoDB needs less loading time due to its use of compression; as expected, its loading time increases as the number of measurements per array is decreased due to less compression.

Comparison with SparkSQL: We next compare with a well-known NoSQL system, SparkSQL. In this experiment we ran Q1 both on VXQuery with the JSONiq rewrite rules and on Spark SQL and we compared their execution times. We used a single node and one core as the setup for both systems. We varied the dataset sizes starting from 400MB up

Data Size (MB)	Loading Time (s)
400	6.3
800	15
1000	40

Table 4.2: Loading Time for Spark SQL

to 1GB. We could not run experiments with larger input files because Spark required more than the available memory space to load such larger datasets.

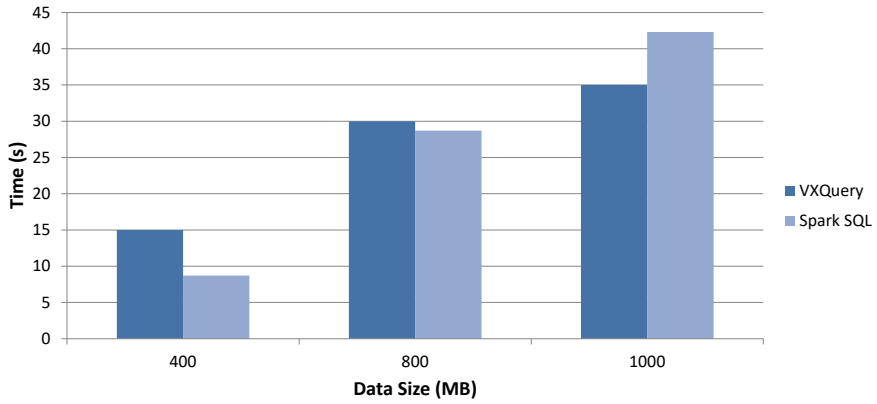


Figure 4.17: Spark SQL vs VXQuery Execution Time for Query Q1 Using Different Data Sizes (MB)

Table 4.2 shows the SparkSQL loading times for the datasets used in this experiment. Figure 4.17 shows the query times for query Q1 for the different data sizes. Note that the VXQuery bar shows the total execution time for each file (which includes the loading and query processing) while the SparkSQL bar corresponds to the query processing time only. VXQuery's total execution time is slower than Spark's query time for small files. The two systems show similar performance for 800MB files. However, as the collection size increases, Spark's behavior starts to deteriorate. For the 1GB dataset our system's overall

Data Size (MB)	Spark Memory (MB)	VXQuery Memory (MB)
400	5650	1690
800	6230	1750
1000	7953	1760

Table 4.3: Data size to system memory in MBs

performance is clearly faster. If one counts also for the file loading time of SparkSQL (the overhead added by loading and converting JSON data to a SQL-like format), the VXQuery performance is faster. While the overhead of the loading phase is not a significant burden for SparkSQL when considering small inputs (400MB) it becomes an important limiting factor even for medium size files (800MB).

We also examined the memory allocated by both systems (Table 4.3). The results show that VXQuery stores only data relevant to the query in memory, as opposed to SparkSQL, which stores everything. For file sizes above 2GB, the memory needs of SparkSQL exceeded the node's available 16GB, so it was unable to load the input data so as to query it.

4.4.4 Cluster Experiments

The goal of this section is to examine the cluster speed-up and scale-up achieved by VXQuery due to our JSONiq rewrite rules and compare it with AsterixDB and MongoDB.

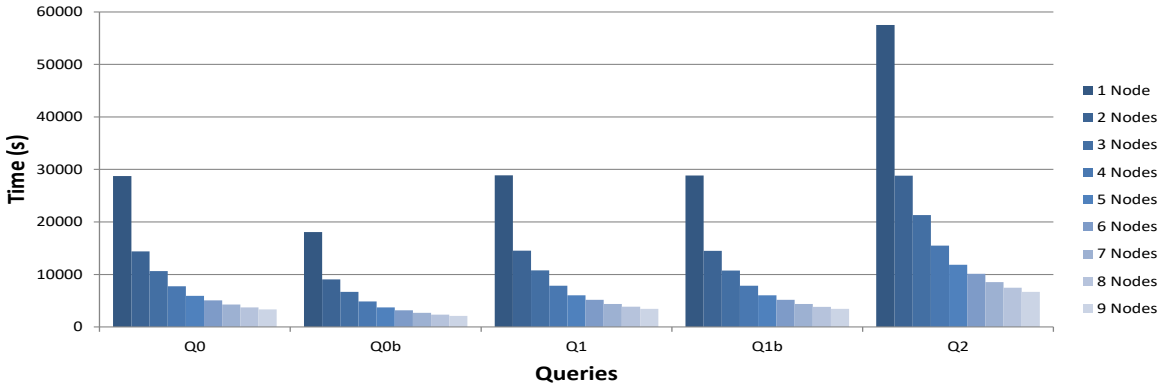


Figure 4.18: VXQuery Cluster Speed-up for all Queries (803GB Dataset)

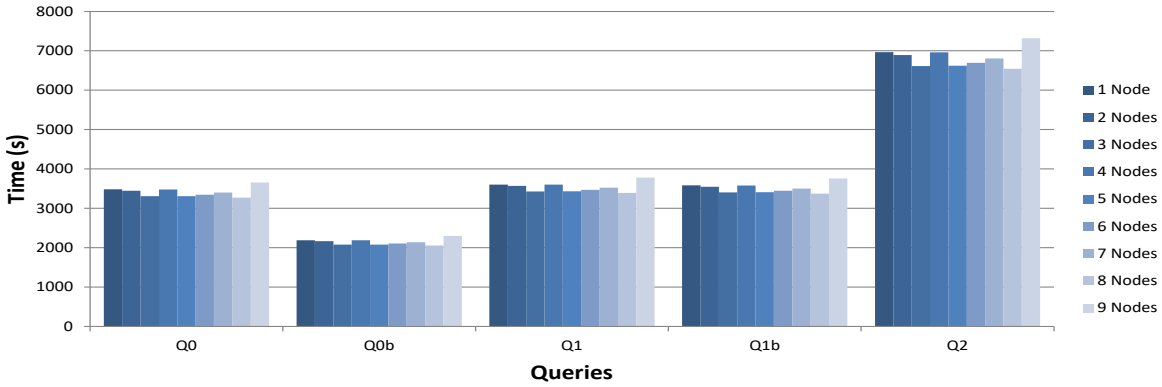


Figure 4.19: VXQuery Cluster Scale Up for all Queries (88GB per Node)

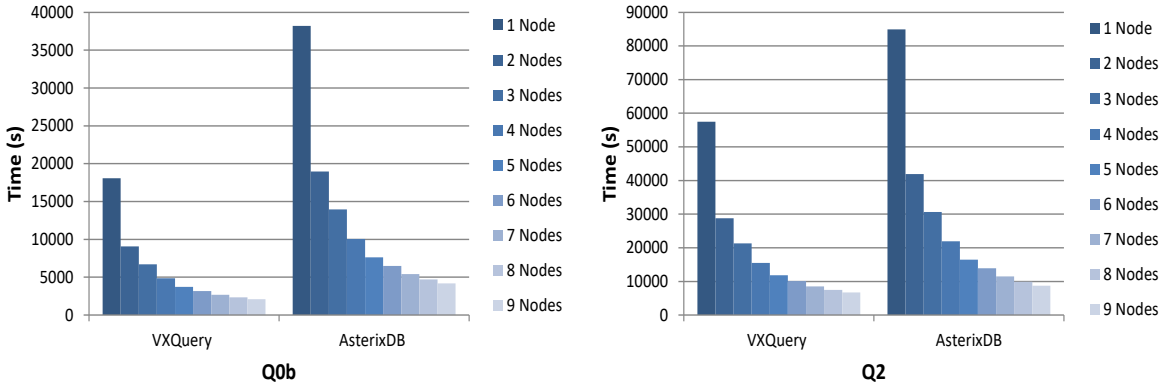


Figure 4.20: VXQuery vs AsterixDB: Cluster Speed-up for Q0b and Q2 (803GB Dataset)

For all the following experiments we used 4 partitions per node which achieves the best execution time as shown in the previous section.

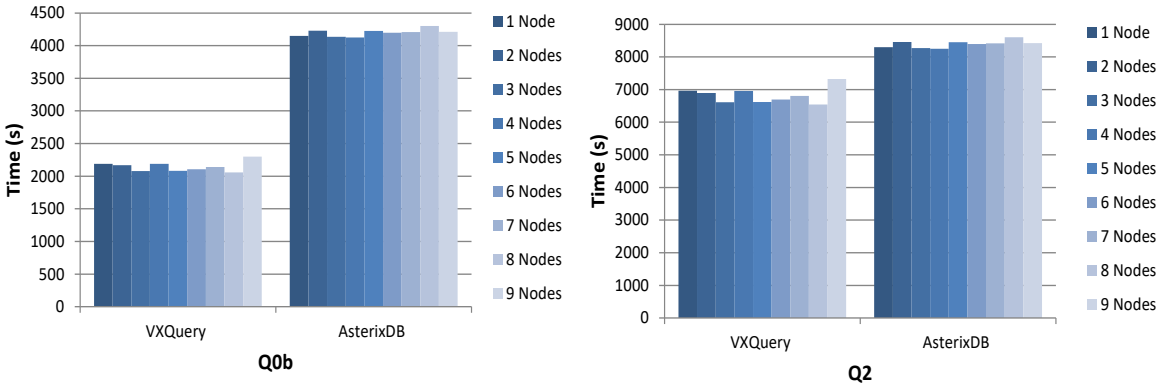


Figure 4.21: VXQuery vs AsterixDB: Cluster Scale-up for Q0b and Q2 (88GB per Node)

To measure the *cluster speed-up* we started with a single node and extended our cluster by one node until it reached to 9 nodes. We used 803GB of JSON weather data which were evenly partitioned among the nodes used in each experiment. This dataset exceeds the available cluster memory. The results for this evaluation are shown in Figure 4.18. We note that in all the cases cluster speed-up is proportional to the number of nodes being used, without depending on the type of the query. We observe that the last query (Q2) takes the most time to execute. This is expected because Q2 is a self join query, which means that it has to process twice the amount of data. On the other hand, for VXQuery, Q0b has the best response time due to its small input search path as described in previous sections.

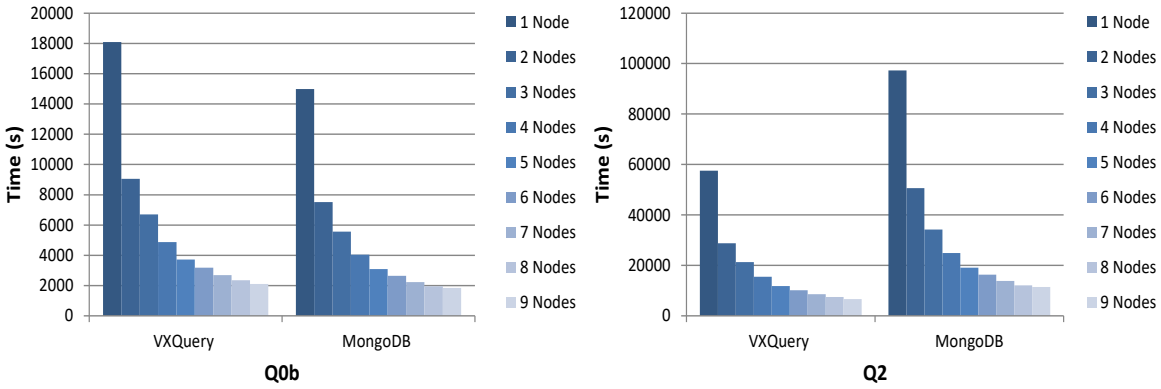


Figure 4.22: VXQuery vs MongoDB: Cluster Speed-up for Q0b and Q2 (803GB Dataset)

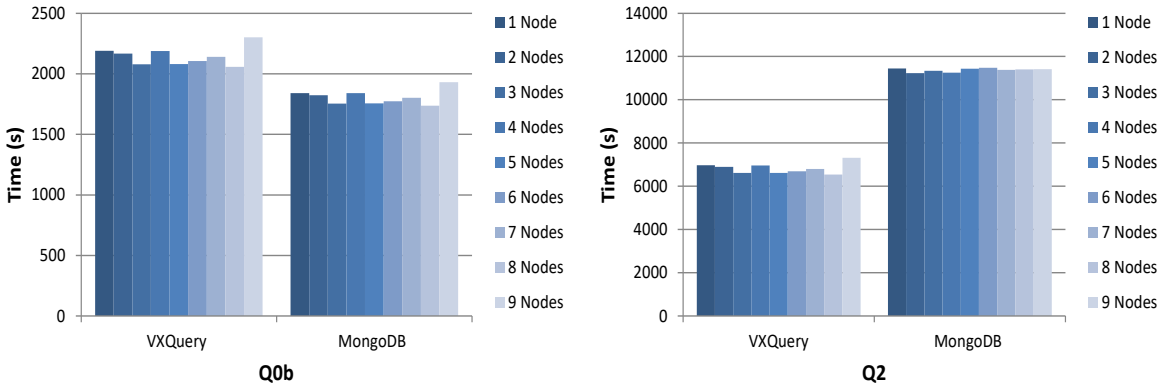


Figure 4.23: VXQuery vs MongoDB: Cluster Scale-up for Q0b and Q2 (88GB per Node)

To show the *scalability* achieved by VXQuery, we started with a dataset of size 88GB which exceeds one node's available memory (8GB). With each additional node added we add four partitions totaling 88GB (hence when using 9 nodes the whole collection is 803GB). The results appear in Figure 4.19. As it can be seen our system achieves very good scale-up performance; the query execution time remains roughly the same, which means that the additional data is processed in roughly the same amount of time.

Comparison with AsterixDB: In the cluster experiments, we compare against AsterixDB (i.e. without loading; each dataset is provided as an external data source). As seen in the single-node experiments, the best performance for AsterixDB is achieved when `"results"` consists of only one measurement; thus we use this structure for the following evaluation.

Following similar reasoning with the single-node comparison, we observe that VXQuery performs better both for speed up (Figure 4.20) and scale up (Figure 4.21), using queries Q0b and Q2 as representative examples.

Comparison with MongoDB: Similarly, we compare against the MongoDB configuration that achieved the best performance in the single-node experiments (i.e., `"results"` contains all monthly measurements). Overall, MongoDB has faster query time for selection queries than VXQuery (Figure 4.22 shows speedup for query Q0b; the Q0 query performed similarly). Since MongoDB performs a compression during the loading phase of the dataset, the dataset stored in the database is much smaller giving an advantage to selection queries. However, VXQuery's execution time for query Q0b is still comparable since its small input search path gives the opportunity for the Pipeline rules to be exploited.

In contrast, VXQuery has a faster execution time than MongoDB on join queries (like Q2).

For this self-join, MongoDB tries to put all the documents that share the same station and date in the same document; thus creating huge documents which exceed the 16MB document size limit causing it to fail. To overcome this problem, we perform an additional step before the actual join. We unwind the `"results"` array and we project only the necessary fields. After that, we perform the actual join on the corresponding attributes (i.e. station, date of measurement).

On the other hand, in VXQuery there is no document size limitation, making VXQuery more efficient in handling large JSON items. Table 4.4 shows the MongoDB loading times per node. This adds a huge overhead to the performance of the overall system and it can be prohibitively large for real-time applications where the dataset may not been known in advance.

Data Size (GB)	Loading Time (sec)
88	9000
803	81000

Table 4.4: Loading Time for MongoDB

Comparison with SparkSQL: As mentioned in the single node experiments SparkSQL could not run experiments with larger input files because of the required memory space to load such larger datasets. Hence we omit multi-node experiments with SparkSQL, since the dataset size will be very small to indicate and difference in the execution time.

4.5 Conclusions and Future Work

In this work we introduced two categories of rewrite rules (path expression and pipelining rules) based on the JSONiq extension to the XQuery specification. We also introduced a third rule category, group-by rules, that apply to both XML and JSON data. The rules enable new opportunities for parallelism by leveraging pipelining; they also reduce the amount of memory required as data is parsed from disk and passed up the pipeline. We integrated these rules into an existing system, the Apache VXQuery query engine. The resulting query engine is the first that can process queries in an efficient and scalable manner on both XML and JSON data.

Through experimentation, we showed that these rules improve performance for various selection, join and aggregation queries. In particular, the pipelining rules improved performance by several orders of magnitude. The system was also shown both to speed-up and scale-up effectively. Moreover, when compared with other systems that can handle

JSON data, VXQuery shows significant advantages. In particular, our system can directly process JSON data efficiently without the need to first load it and transform it to an internal data model.

We are currently working on supporting indexing over both types of data (XML and JSON). Indexing presents a significant challenge, as there is no simple way to decide the level at which an object could be indexed; indexing will further improve the system's performance since the searched data volume will be significantly reduced. All of the code for our JSONiq extension is available through the Apache VXQuery site [9] and it will be included in the next Apache VXQuery release. Furthermore, we plan to add the proposed path and pipelining rules directly to AsterixDB given that it shares the same infrastructure (Algebricks and Hyracks) with VXQuery.

Chapter 5

Conclusion and Future Work

In recent years there has been a huge influx on the volume of data generated by social media, IoT devices, etc. This trend combined with the increasing demand in complex analytics over those data, puts significant pressure on the processing capabilities of traditional database systems. In this dissertation, we concentrated on tackling the challenges related to query optimization in modern data-intensive systems.

In Chapter 3, we focus on runtime dynamic optimization in BDMS. Specifically, we execute a submitted query in stages (re-optimization points). The intermediate results generated by each stage are used to collect updated statistics and re-optimize the remaining query. Moreover, we handle queries in which base datasets are accessed through complex predicates (i.e multiple predicates with undetected correlations, external variables and UDFs). Following this dynamic approach, we acquire accurate statistics for post-filter application datasets and thereby making more informed decisions for join order and algorithm. We integrate the dynamic optimization approach in a BDMS, namely AsterixDB which is

optimized to execute joins in a pipeline. We evaluated the runtime dynamic optimization against static cost-based optimization, the default AsterixDB query execution approach and a state-of-the-art approach. Our results showed that even though the dynamic optimization blocks the pipelining feature and introduces an overhead by producing intermediate results, it still gives almost always the best performance. Our dynamic approach generates the best execution plans which in the big data era is very important, as a small error in join ordering can have drastic consequences on query performance.

In Chapter 4, we concentrated on querying efficiently JSON data. In particular, we presented a collection of rewrite rules integrated into VXQuery, that promote the efficient processing of deeply nested JSON data. These rules leverage on some key features of the JSONiq language to merge together the processing of multiple query operators. They, also, exploit the pipelining feature of VXQuery to promote parallelism. We have carried out an extensive experimental evaluation that verified that our heuristic rules improve tremendously the performance of VXQuery against deeply nested JSON data. Moreover, we compared against other systems that can process JSON data and the results show that VXQuery can query JSON data very efficiently and directly without the need of any transformation to internal data formats.

In future research, we would like to optimize the dynamic optimization approach further by exploring the benefits of learning from past query executions and thus minimize the overhead of multiple re-optimizations. Another promising direction is to integrate more costly UDFs in our dynamic optimization approach, so that more appropriate decisions about UDF evaluation order (e.g pull up/push down). Currently our dynamic optimizer orders

grouping and ordering operators after the execution of joins. We would like to extend the dynamic optimizer's implementation to handle the aforementioned operators and thus offer a more broad spectrum of efficient execution plans. One could also exploit the materialization points introduced by the runtime dynamic optimization approach in fault-tolerance scenarios to avoid failure of long running queries. Finally, we wish to add the path and pipelining rules proposed in Chapter 4 directly to AsterixDB, given that it shares the same infrastructure (Algebricks and Hyracks) with VXQuery, to see the benefits brought by them in processing very nested JSON data.

Bibliography

[1] Ibm knowledge center. https://www.ibm.com/support/knowledgecenter/.

[2] NOAA weather data.

[3] Pascal xquery engine. http://www.benibela.de.

[4] Google gson. https://github.com/google/gson, 2012.

[5] Jackson project. https://github.com/FasterXML/jackson, 2012.

[6] Zorba. http://www.zorba.io/home, 2013.

[7] Apache Parquet. https://parquet.apache.org/, 2014.

[8] Rapidjson. http://rapidjson.org/, 2015.

[9] Apache vxquery. `http://vxquery.apache.org`, 2016.

[10] Apache Drill. https://drill.apache.org/, 2017.

[11] Apache Spark. https://spark.apache.org/, 2017.

[12] JSONiq Extension to XQuery. http://www.jsoniq.org/docs/JSONiqExtensionToXQuery/html-single/index.html, 2017.

[13] JSONiq language. http://www.jsoniq.org/docs/JSONiq/html-single/index.html, 2017.

[14] MongoDB. https://www.mongodb.com/, 2017.

[15] Postgres-XL. https://www.postgres-xl.org/, 2017.

[16] PostgreSQL. https://www.postgresql.org/, 2017.

[17] *AsterixDB*, 2020.

[18] *TPC*, 2020.

[19] *TPCDS*, 2020.

[20] *TPCH*, 2020.

[21] Ildar Absalyamov. *Query Processing and Cardinality Estimation in Modern Database Systems*. PhD thesis, UNIVERSITY OF CALIFORNIA RIVERSIDE, 2018.

[22] Ildar Absalyamov, Michael J Carey, and Vassilis J Tsotras. Lightweight cardinality estimation in lsm-based systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 841–855, 2018.

[23] Sameer Agarwal, Srikanth Kandula, Nico Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. Reoptimizing data parallel computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 281–294, 2012.

[24] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, et al. Asterixdb: A scalable, open source bdms. *arXiv preprint arXiv:1407.0454*, 2014.

[25] Michael Armbrust et al. Spark SQL: Relational data processing in Spark. In *Proceedings of ACM SIGMOD Conference*, pages 1383–1394, 2015.

[26] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. System r: relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[27] Ron Avnur and Joseph M Hellerstein. Eddies: Continuously adaptive query processing. In *ACM sigmod record*, volume 29, pages 261–272. ACM, 2000.

[28] Shivnath Babu, Pedro Bizarro, and David DeWitt. Proactive re-optimization. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 107–118. ACM, 2005.

[29] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 119–130, 2010.

[30] Kevin S Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh, Carl-Christian Kanne, Fatma Ozcan, and Eugene J Shekita. Jaql: A scripting language for large scale semistructured data analysis. *Proceedings of the VLDB Endowment*, 4(12):1272–1283, 2011.

[31] Vinayak Borkar, Yingyi Bu, E Preston Carman Jr, Nicola Onose, Till Westmann, Pouria Pirzadeh, Michael J Carey, and Vassilis J Tsotras. Algebricks: a data model-agnostic compiler backend for big data languages. In *Proceedings of 6th ACM Symposium on Cloud Computing*, pages 422–433, 2015.

[32] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *27th International Conference on Data Engineering*, pages 1151–1162, 2011.

[33] Nicolas Bruno, Sapna Jain, and Jingren Zhou. Continuous cloud-scale query optimization and processing. *Proceedings of the VLDB Endowment*, 6(11):961–972, 2013.

[34] E Preston Carman, Till Westmann, Vinayak R Borkar, Michael J Carey, and Vassilis J Tsotras. A scalable parallel xquery processor. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 164–173. IEEE, 2015.

[35] E Preston Carman, Till Westmann, Vinayak R Borkar, Michael J Carey, and Vassilis J Tsotras. A scalable parallel XQuery processor. In *IEEE International Conference on Big Data*, pages 164–173, 2015.

[36] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment*, 8(4):401–412, 2014.

[37] Craig Chasseur, Yinan Li, and Jignesh M Patel. Enabling json document stores in relational systems. In *WebDB*, volume 13, pages 14–15, 2013.

[38] Surajit Chaudhuri and Kyuseok Shim. Optimization of queries with user-defined predicates. *ACM Transactions on Database Systems (TODS)*, 24(2):177–228, 1999.

[39] Jack Chen, Samir Jindel, Robert Walzer, Rajkumar Sen, Nika Jimsheleishvilli, and Michael Andrews. The memsql query optimizer: A modern optimizer for real-time analytics in a distributed database. *Proceedings of the VLDB Endowment*, 9(13):1401–1412, 2016.

[40] Amol Deshpande, Zachary Ives, Vijayshankar Raman, et al. Adaptive query processing. *Foundations and Trends® in Databases*, 1(1):1–140, 2007.

[41] Yuan Yu Michael Isard Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, and Pradeep Kumar Gunda Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. *Proc. LSDS-IR*, 8, 2009.

[42] Gabriel Filios, Sotiris Nikoletseas, Christina Pavlopoulou, Maria Rapti, and Sébastien Ziegler. Hierarchical algorithm for daily activity recognition via smartphone sensors. In *2nd IEEE World Forum on Internet of Things (WF-IoT)*, pages 381–386, 2015.

[43] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. 2007.

[44] Goetz Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[45] Goetz Graefe and David J DeWitt. *The EXODUS optimizer generator*, volume 16. ACM, 1987.

[46] Goetz Graefe and William J McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of IEEE 9th International Conference on Data Engineering*, pages 209–218. IEEE, 1993.

[47] Michael Greenwald and Sanjeev Khanna. Space-efficient online computation of quantile summaries. *ACM SIGMOD Record*, 30(2):58–66, 2001.

[48] Christian Grün, Sebastian Gath, Alexander Holupirek, and Marc H Scholl. Xquery full text implementation in basex. In *International XML Database Symposium*, pages 114–128. Springer, 2009.

[49] Joseph M Hellerstein. Optimization techniques for queries with expensive methods. *ACM Transactions on Database Systems (TODS)*, 23(2):113–157, 1998.

[50] Ihab F Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. Cords: automatic discovery of correlations and soft functional dependencies. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 647–658, 2004.

[51] Yannis E Ioannidis and Stavros Christodoulakis. On the propagation of errors in the size of join results. In *Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, pages 268–277, 1991.

[52] Navin Kabra and David J DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *ACM SIGMOD Record*, volume 27, pages 106–117. ACM, 1998.

[53] Konstantinos Karanasos, Andrey Balmin, Marcel Kutsch, Fatma Ozcan, Vuk Ercegovac, Chunyang Xia, and Jesse Jackson. Dynamically optimizing queries over large scale data platforms. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 943–954. ACM, 2014.

[54] Qifa Ke, Michael Isard, and Yuan Yu. Optimus: a dynamic rewriting framework for data-parallel execution plans. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 15–28. ACM, 2013.

[55] Felix N Kogan. Droughts of the late 1980s in the united states as derived from noaa polar-orbiting satellite data. *Bulletin of the American Meteorological Society*, 76(5):655–668, 1995.

[56] Shamanth Kumar, Fred Morstatter, and Huan Liu. *Twitter Data Analytics*. Springer Publishing Company, 2013.

[57] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. The vertica analytic database: C-store 7 years later. *arXiv preprint arXiv:1208.4173*, 2012.

[58] Yinan Li, Nikos R Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. Mison: a fast JSON parser for data analytics. In *Proceedings of the VLDB Endowment*, volume 10, pages 1118–1129, 2017.

[59] Youfu Li, Mingda Li, Ling Ding, and Matteo Interlandi. Rios: Runtime integrated optimizer for spark. In *SoCC*, pages 275–287, 2018.

[60] Jimmy Lin and Dmitriy Ryaboy. Scaling big data mining infrastructure: the twitter experience. *ACM SIGKDD Explorations Newsletter*, 14(2):6–19, 2013.

[61] Zhen Hua Liu, Beda Hammerschmidt, and Doug McMahon. JSON data management: supporting schema-less development in RDBMS. In *Proceedings of ACM SIGMOD Conference*, pages 1247–1258, 2014.

[62] Ioana Manolescu, Daniela Florescu, and Donald Kossmann. Answering xml queries on heterogeneous data sources. In *VLDB*, volume 1, pages 241–250, 2001.

[63] Jason McHugh and Jennifer Widom. Query optimization for XML. *Proceedings of the 25th VLDB Conference*, 1999.

[64] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[65] Sriram Mohan, Arijit Sengupta, and Yuqing Wu. Access control for xml: a dynamic query rewriting approach. In *Proceedings of the 14th ACM CIKM Conference*, pages 251–252, 2005.

[66] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. Comparison of JSON and XML data interchange formats: a case study. *22nd Intl. Conference on Computer Applications in Industry and Engineering (CAINE)*, pages 157–162, 2009.

[67] Leonard Richardson and Sam Ruby. *RESTful web services*. " O'Reilly Media, Inc.", 2008.

[68] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34, 1979.

[69] Mohamed A Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, et al. Orca: a modular query optimizer architecture for big data. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 337–348, 2014.

[70] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. Leo-db2's learning optimizer. In *VLDB*, volume 1, pages 19–28, 2001.

[71] Daniel Tahara, Thaddeus Diamond, and Daniel J Abadi. Sinew: a sql system for multi-structured data. In *Proceedings of ACM SIGMOD Conference*, pages 815–826. ACM, 2014.

[72] Immanuel Trummer and Christoph Koch. Parallelizing query optimization on shared-nothing architectures. *Proceedings of the VLDB Endowment*, 9(9):660–671, 2016.

[73] Jean-Yves Vion-Dury. Xpath on left and right sides of rules: toward compact xml tree rewriting through node patterns. In *Proceedings of the 2003 ACM symposium on Document Engineering*, pages 19–25. ACM, 2003.

[74] Eugene Wong and Karel Youssefi. Decomposition—a strategy for query processing. *ACM Transactions on Database Systems (TODS)*, 1(3):223–241, 1976.

[75] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*, pages 13–24, 2013.

[76] Cong Yu and Lucian Popa. Constraint-based xml query rewriting for data integration. In *SIGMOD*, pages 371–382. ACM, 2004.

[77] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 15–28, 2012.

[78] Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M Procopiuc, and Divesh Srivastava. Automatic discovery of attributes in relational databases. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 109–120, 2011.

[79] Xin Zhang, Mukesh Mulchandani, Steffen Christ, Brian Murphy, and Elke A Rundensteiner. Rainbow: mapping-driven xquery processing system. In *Proceedings of ACM SIGMOD Conference*, pages 614–614, 2002.

[80] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. Scope: parallel databases meet mapreduce. *The VLDB Journal*, 21(5):611–636, 2012.