UNIVERSITY OF CALIFORNIA,
IRVINE

Result Distribution in Big Data Systems

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Science

by

Madhusdan Cheelangi

Thesis Committee:
Prof. Michael J. Carey, Chair
Prof. Chen Li
Prof. Cristina Lopes

2013

To Melange open source software and to my friends Lennard de Rijk, Sverre Rabbelier, Carol Smith, Mario Ferraro and Daniel Hans.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to thank Vinayak Borkar for helping me through out this project from the initial design phases to the final phases of planning and executing the experiments by giving his valuable ideas and inputs at each phase of the project. I would also like to thank him for helping me with understanding the Hyracks codebase. I would also like to thank Yingyi Bu for helping me understand the code of various components of AsterixDB and also for helping me profile my code. I would like to thank Raman Grover for helping me understand how AsterixDB delivered the results to the clients before this effort. I would like to thank Sattam Alsubaiee for timely code reviews and his feedback on result distribution, Zachary Heilbron and Young-Seok Kim for their valuable inputs on how to design concurrent datastructures, Pouria Pirzadeh and Jian Wen for their inputs on debugging the system. I would also like to thank Khurram Faraaz for porting the AsterixDB's test framework to the new REST API that was introduced as part of this project that enabled me to test my result distribution framework. I would also like to thank him for being a friend-in-need through out the thesis writing process.

I would like to thank my thesis committee members Prof. Chen Li and Prof. Cristina Lopes for reviewing my thesis.

ASTERIX project is supported by an eBay matching grant, one Facebook Fellowship Award, the NSF Awards No. IIS-0910989, IIS-0910859, and IIS-0910820, a UC Discovery grant, three Yahoo! Key Scientific Challenge Awards, and generous industrial gifts from Google, HTC, Microsoft and Oracle Labs. I would like to thank all our sponsors for supporting our project.

I would like to thank my mom and my friend Krishna Bharadwaj for keeping me motivated through out the duration of the project and the thesis writing process, checking with me every single day about my progress, and for patiently listening to whatever I had to say.

As always, most important things towards the end. I would like to thank my advisor Prof. Michael J. Carey for his invaluable feedback and comments through out the course of this project and on the thesis. I would also like to thank him for his timely reviews of the thesis. He has been a source of inspiration. Anything I have written here or I am going to write further to thank him is going to be an understatement, I would just like to close this paragraph by thanking him for giving me an opportunity to work with him and with the ASTERIX team.

ABSTRACT OF THE THESIS

Result Distribution in Big Data Systems

By

Madhusdan Cheelangi

Master of Science in Computer Science

University of California, Irvine, 2013

Prof. Michael J. Carey, Chair

We are building a Big Data Management System (BDMS) called **AsterixDB** at UCI. Since AsterixDB is designed to operate on large volumes of data, the results for its queries can be potentially very large, and AsterixDB is also designed to operate under high concurency workloads. As a result, we need a specialized mechanism to manage these large volumes of query results and deliver them to the clients. In this thesis, we present an architecture and an implementation of a new result distribution framework that is capable of handling large volumes of results under high concurency workloads. We present the various components of this result distribution framework and show how they interact with each other to manage large volumes of query results and deliver them to clients. We also discuss various result distribution policies that are possible with our framework and compare their performance through experiments.

We have implemented a REST-like HTTP client interface on top of the result distribution framework to allow clients to submit queries and obtain their results. This client interface provides two modes for clients to choose from to read their query results: synchronous mode and asynchronous mode. In synchronous mode, query results are delivered to a client as a direct response to its query within the same request-response cycle. In asynchronous mode, a query handle is returned instead to the client as a response to its query. The client can

store the handle and send another request later, including the query handle, to read the result for the query whenever it wants. The architectural support for these two modes is also described in this thesis. We believe that the result distribution framework, combined with this client interface, successfully meets the result management demands of AsterixDB.

Chapter 1

Introduction

Traditionally, Database Management Systems (DBMS) have been responsible for both storing data and executing user queries. In order to execute user queries, a DBMS needs a powerful runtime system. Such a runtime system typically performs a variety of tasks. It compiles the given query to generate the plans to be executed. It optimizes these plans and then executes them. It finally delivers the retrieved results to the user. In the case of database systems that are designed to run on a single computer, all these tasks are executed on the same computer (server) where the database system is installed.

Due to the unprecedented growth of data in the last few decades, it is becoming increasingly difficult to build single-computer database systems that can store and operate on such vast amounts of data. The sheer magnitude of data has led to the coinage of the new term, "Big Data" for referring to such vast amounts of data. The history of Big Data systems dates back to early 1980's [5]. The need of the enterprises to store and analyze their historical business data led to the development of parallel databases on "shared-nothing" architecture. The architecture of shared-nothing parallel database systems involved a networked cluster of independent machines that communicated via message passing. These systems partitioned

the data for storage and query execution across individual machines of a shared-nothing cluster. The advent of World-Wide Web in 1990's and the challenges posed by the Web-scale operations of Internet companies led to the current Big Data revolution. Google's introduction of Google File System and Bigtable for storing Web-scale data and the introduction of MapReduce programming model for processing such large volumes of data led to the creation of its open-source equivalent Hadoop ecosystem [6]. The Hadoop stack of Big Data systems are also designed to run on a collection of computers that are interconnected via a network. However, unlike parallel database systems, the Hadoop stack is composed of a number of non-monolithic components that provide support for replication, automatic data placement and load balancing. It is also capable of scheduling large jobs in smaller chunks. On the other hand, the Hadoop stack only provides unary operators - map and reduce for all the operations while traditional parallel database systems generally come with a rich set of operators. However, in both the architectures, the computers on which a Big Data system runs are referred to as its nodes, and a database system that operates on Big Data distributes the data to be stored and processed among these nodes. Hence, the database system's runtime must be capable of executing the query plans beginning on the nodes where the data is stored and must aggregate the computed final result set before delivering it to the user.

Due to the large volume of data that these Big Data systems operate on, the results to the queries can also be very large. In addition, because of the scale at which these systems are deployed, they may have to answer tens, hundreds or even thousands of queries concurrently. Therefore, these systems should be capable of handling large volumes of results. We are building a new, open-source Big Data Management System (BDMS), AsterixDB, at UCI with the goal of bringing the best of both the parallel databases world and the Hadoop world together [2]. Being a Big Data system, AsterixDB needed a mechanism to handle large volumes of results. Hence, we built a new mechanism for aggregating and distributing results in AsterixDB. Also, not all the clients have the need to consume the results for the queries as the system generates the results. Especially for long running queries, it may be

desirable to have the clients submit the query and read the results later when they are ready to be consumed instead of having the clients wait for the results to appear while holding the resources. Hence, in addition to synchronous queries where clients read the results as they are generated by the system, we also implemented the support for asynchronous queries in AsterixDB, where the clients can read the results when they want to. In the following chapters, we will discuss the architecture, implementation and performance of the result distribution mechanism for both synchronous and asynchronous queries built for use in AsterixDB.

Chapter 2

Motivation

Database systems provide a query interface to retrieve the stored data. The queries return results that contain zero or more records. Big Data systems are designed to store and operate on very large volumes of data, typically in the order of terabytes or petabytes. Consequently, the number of records in the results for the queries can be very large too. Furthermore, Big Data systems are also designed for high concurrency workloads, i.e. they are often intended to execute several tens or hundreds of queries concurrently, especially when the system's hardware base is a large cluster with hundreds or even thousands of nodes. In such a BDMS setting we do not want client consumption rates to have any impact on the system performance.

In a traditional Relational DBMS, clients' consumption throttles query execution and hence the rate of result production. While the system executes the query, locks may be held on database tables and other system resources. In a BDMS, we think this will have an undesirable effect on the system's performance since resources could be tied up by queries whose execution rates are dictated by client consumption rates, with tens or thousands of concurrently executing queries competing for resources. Hence, we need a mechanism to

temporarily store query results somewhere until they are consumed by clients in order to decouple the query execution and result production pipeline from the clients' result consumption pipelines.

As queries are executed in the system, the results produced by these queries must be aggregated (i.e., gathered into a single logical result stream) before they are delivered back to the clients that submitted the queries. This naturally leads to the question of how and when and where these results should be aggregated before they are handed over to the clients. A related question is, where should the results be stored if the clients' result consumption rates are slower than the rate at which the results are produced? We define this problem of storing and aggregating the results of the queries in a database system as the **"Result Distribution Problem"**. This thesis describes the principles and architecture underlying such a framework for result distribution as implemented in AsterixDB.

AsterixDB is a database system designed to operate on Big Data. The ability of AsterixDB to concurrently execute a large number of queries combined with large result set sizes calls for a specialized mechanism to handle the query results before they are handed over to the clients. As queries are executed by AsterixDB, the results for each query are incrementally generated in memory. However, memory is a critical resource. Every operation in the system, including execution of queries, storing system state, etc. requires memory. Hence, there is only a limited amount of memory that can be budgeted towards storing results. Due to the potentially large sizes of result sets, it is not sensible to store all results in memory; some results should be stored on disk. However, disk I/Os are several orders of magnitude slower than main memory access, and every result that must be stored on disk adds an overhead of two disk I/Os: one to write the result to the disk for storage and another to read the result back from the disk later to deliver it to the client. Another consideration is that different clients can have different consumption rates, so it may be more beneficial to store the results of some queries in memory than others. Also, even though AsterixDB is a Big

Data system and queries can generate large results, not all queries will necessarily generate large results. For example, a simple primary key lookup query at the most generates just one record as its result. Therefore, BDMS query result cardinalities are likely to vary from a single record to millions of records, i.e. the result data sizes will vary from a few bytes to hundreds of megabytes. Again, we may have to decide storing which portitons of which results in memory over the others is beneficial.

AsterixDB is a parallel Big Data Management System which is designed to run on shared-nothing clusters. Each node in such a cluster independently executes the tasks assigned to it. A query to AsterixDB is compiled into a job and these jobs are divided into tasks to be executed by the nodes in the cluster. Data flows through these nodes as the query executes. Hence, if the result for a query is generated at multiple nodes, their results must all be aggregated before delivering the query result to the client. This leads to the question, should we store the entire result set corresponding to a query on a single node before delivering it to the client? Or is it possible to leverage the advantages of the distributed nature of shared-nothing clusters to store the result on the nodes where they are produced and read them from those nodes directly while delivering the result to the client?

The result distribution problem, thus, provides an opportunity for implementing several different policies for storing and delivering the results to the client. Since every query produces zero or more records as its result, the result distribution framework is a critical piece of a Big Data management system. Hence, the performance of the system depends, in part, on the choice of its result distribution policies. Therefore it is important to study the effects of these policies on the overall system performance. While several different combinations of policies are possible, this thesis discusses only a few of the possible result distribution policies that are implemented in AsterixDB.

In addition to the result distribution framework provided by AsterixDB, clients need an interface to talk to AsterixDB, i.e., an interface to submit queries and to get their results

back. There are multiple ways in which such a client interface can be implemented. The client interface can be as simple as a command line interface through which queries can be submitted and results read, or it can be an API in some particular programming language. We wanted to provide an API that was programming language agnostic. One such mechanism that allows for the provision of a programming language agnostic API is REpresentationl State Transfer (REST). Hence, this thesis also discusses the implementation of the REST API in AsterixDB that allows clients to submit queries and read the results.

In summary, the contributions of this thesis include: an architecture for result distribution in Big Data management systems, an implementation of the result distribution framework in AsterixDB, a study of a few possible result distribution policies, and a REST API for clients to interact with AsterixDB.

Chapter 3

Architecture

3.1 Overview of the Architecture of AsterixDB

Let us begin with an overview of the architecture of AsterixDB before delving into the architecture of the result distribution framework. AsterixDB is a parallel Big Data Management System (BDMS) designed to run on shared-nothing, commodity clusters [4]. AsterixDB has its own data model called the Asterix Data Model (ADM) for storing and representing the data it operates on. AsterixDB provides a query interface through its own query language called the Asterix Query Language (AQL). AsterixDB consists of various components that interact with each other to store and retrieve data. These components include the AQL parser, the Algebricks algebraic query compiler and optimizer, and Hyracks, a data-parallel runtime system. Hyracks in turn consists of a Cluster Controller that coordinates the activities of the cluster and a set of Node Controllers running on each node of the cluster.

A client submits its query to AsterixDB through AsterixDB's client interface. This query is then passed to the AQL Parser to build a syntax tree. The syntax tree is then converted into a logical query plan by Algebricks. A rule-based query optimizer in Algebricks then rewrites

the logical query plan to obtain the optimized query plan. Algebricks then generates a Hyracks job description for the query plan to be executed on the cluster. The Hyracks job description is then submitted to the Cluster Controller for execution.

At the Hyracks level [3], the Cluster Controller processes the job description and generates a set of tasks to be executed by the Node Controllers in order to execute the query. It sends each task description to the corresponding Node Controller chosen to execute the task. A task description consists of a sequence of operators that are to be executed by the Node Controller in order to execute that task. The data stored in AsterixDB is partitioned, and the data corresponding to each partition physically resides on a node in the cluster where one of the Node Controllers runs. A Node Controller receives a task description and executes the sequence of operators defined in the task description on the data partition(s) that resides locally on that node in the cluster. The output of each operator is pushed as an input to the next operator as specified in the task description. Node Controllers may have to exchange data between them in order to complete job execution. Once all the operators specified in the task description given to a Node Controller are executed, its output is pushed as input to the first operator in the task description of another Node Controller that is waiting to receive data from the previous Node Controller. This sequence continues until the final operator in the query plan which generates the results for the query is executed. The generated result is finally delivered to the result distribution framework which takes the responsibility of delivering the result to the client which sent the query. The unit of data communication through out the system is a frame containing a number of data objects.

The Hyracks Cluster Controller performs various coordination activities in addition to generating task descriptions from a given job. It maintains a registry of all the Node Controllers that are active in the system, keeping track of the status of each Node Controller using a heartbeat mechanism. It also tracks the progress of a job as each Node Controller executes the tasks that form the job. The Cluster Controller provides an interface for clients to com-

municate with the system, i.e., to provide the job description, to start and stop jobs, to retrieve job status, and so on. Node Controllers, on the other hand, just register themselves with the Cluster Controller when they join the cluster, report their status by sending a heartbeat signal at regular intervals to the Cluster Controller, and execute the tasks that the Cluster Controller assigns to them.

3.2 Result Distribution Architecture

AsterixDB's AsterixDB API provides an interface through which clients can submit AQL statements. The details of the architecture of this client interface will be described later in this chapter. For now, it is important to know that the AsterixDB API provides two different modes for queries:

1. Synchronous mode, in which a query is submitted as part of the request and the result is sent back as a response to the request. The client should remain connected to AsterixDB until the query is executed by the system and the entire result set has been sent back as a response. If the client loses the connection in between, the query has to be resubmitted as a new one. Queries submitted in synchronous mode are called synchronous queries.

2. Asychronous mode, in which a query is submitted as part of the request and a handle to the query is sent back as a response to the request. The client can store the handle and can later use that handle either in the query status API to check the status of the query or in the query result API to fetch the results for the query. Asynchronous mode is beneficial for long running queries because the client need not keep its connection to AsterixDB active after sending the query. It can connect again later whenever it wants to check the status of the query or read the result. Queries submitted in asynchronous

mode are called asynchronous queries.

3.2.1 Synchronous Queries

The architecture of result distribution for both synchronous and asynchronous queries is similar with some minor differences in the user-facing components. Therefore, we will first discuss the architecture for synchronous queries and then extend it to asynchronous queries in the next section.

Figure 3.1 depicts the architecture of result distribution for synchronous queries. The user-facing component of AsterixDB is a query execution and result distribution server that listens for requests from the client. A client sends its query to the AsterixDB API server as a request (1 in Figure 3.4 - parenthesized numbers refer to the corresponding step in the diagram) and waits for a response containing the result for the query. The server's Query Handler extracts the query statement from the request and calls the AQLParser with the extracted statement as an argument (2). The AQLParser parses the query statement, builds a syntax tree from it, and returns the syntax tree to the Query Handler. The Query Handler then calls the Algebricks query compiler with the syntax tree as a call argument (3).

The query goes through three layers of transformations in Algebricks. In the first phase, the syntax tree is transformed into a logical plan. In the second phase, the rule-based optimizer in Algebricks rewrites the logical plan to produce an optimized logical plan. In the final phase, the optimized logical plan is converted into a physical plan that guides the generation of a Hyracks job description. This Hyracks job description is then returned to the result distribution server's Query Handler.

The Query Handler submits the resulting Hyracks job description to the Hyracks Cluster Controller (4) which then returns the JobId for that job. The Query Handler then sends

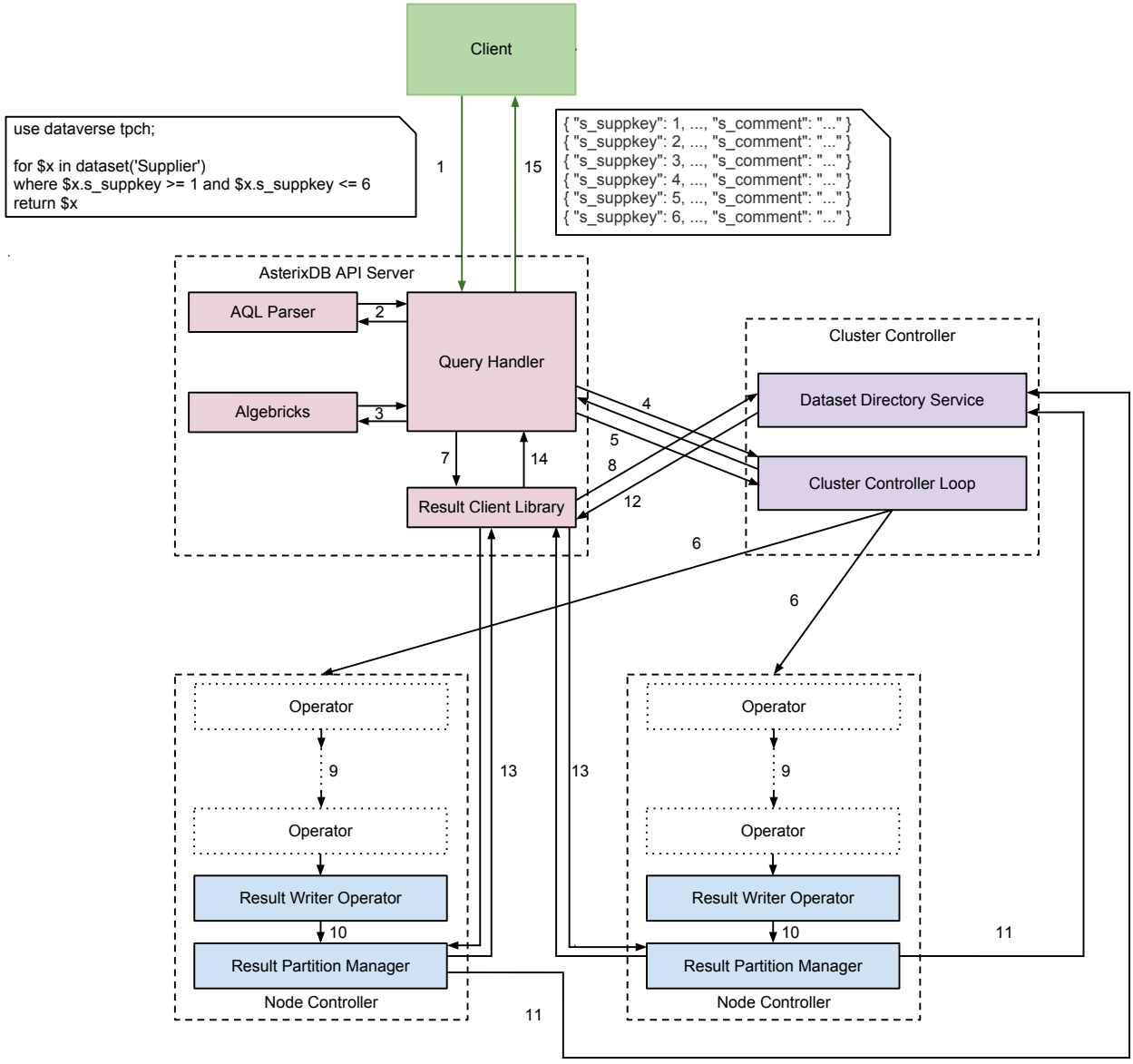


Figure 3.1: Result Distribution Architecture: Synchronous Queries

a start job command with the JobId to the Cluster Controller (5). The Cluster Controller interprets the received job description, determines which Node Controllers are required to execute the job, checks their availability, and then builds the task description based on these information. It sends a start task command to each Node Controller that must execute a task along with the description of the task (6).

While job execution proceeds at the Cluster Controller and Node Controllers, the Query

Handler calls the Result Client Library to start to read the results for the job (7). This call happens immediately after sending the start job command. The Result Client Library (sometimes just referred to as the Result Client) sends a request to the Cluster Controller to obtain the location of the Directory Service. The Directory Service is a service that stores Ids of jobs along with their status and the addresses of all the locations where pieces of the result for the jobs are being generated. Given the Directory Service's location, the Result Client Library makes a Remote Procedure Call to the Directory Service to obtain the result locations corresponding to the JobId (8). If the Directory Service contains an entry for the JobId, it returns all the locations that it is aware of to the Result Client Library. Otherwise, it blocks the call until it learns about the result locations for the requested JobId or the job fails.

On the Node Controllers, as the tasks continue to execute, they eventually reach the last operator in the sequence of operators (9). This last operator is called the Result Writer operator and is responsible for handling the job's result at each Node Controller. Since the job execution is distributed across multiple Node Controllers, depending on the nature of the query, it is possible (likely) that result is generated at more than one Node Controller for the job. Each such portion of the result that is generated at a Node Controller is referred to as the partition of the result and is assigned a PartitionId.

Figure 3.2 depicts the flow of data as frames through the operators at the Node Controllers. The first thing that the Result Writer operator at each Node Controller does when task execution reaches this operator is to register itself, along with the JobId and the PartitionId for which it is generating the result, with the Result Partition Manager running on its Node Controller (10). The Result Partition Manager in turn registers the JobId and the PartitionId along with its location with the Directory Service (11). The Result Partition Manager then creates an instance of a Partition Result Writer for the Job's partition and returns the Partition Result Writer instance to the Result Writer Operator.

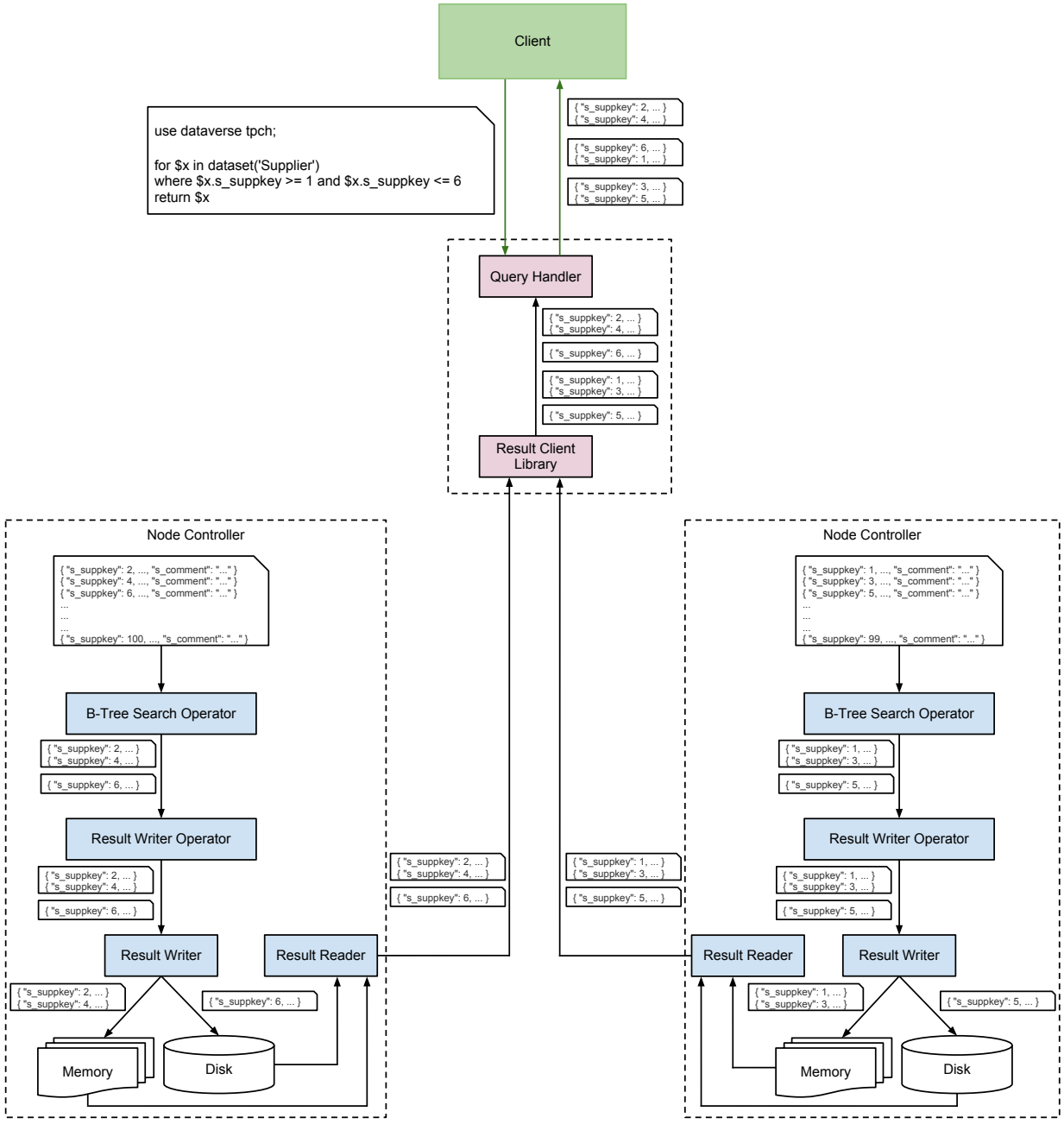


Figure 3.2: Dataflow Through Operators For A Range Query (Primary Key)

In addition to the Result Partition Manager, per-partition instances of Result Writer operator and per-partition instances of Partition Result Writer, the result distribution framework also consists of a single Partition Memory Manager, and per-partition instances of Partition Result Reader and Result State at each Node Controller. Figure 3.3 depicts the components of the result distribution framework at each Node Controller. Result Partition Manager is

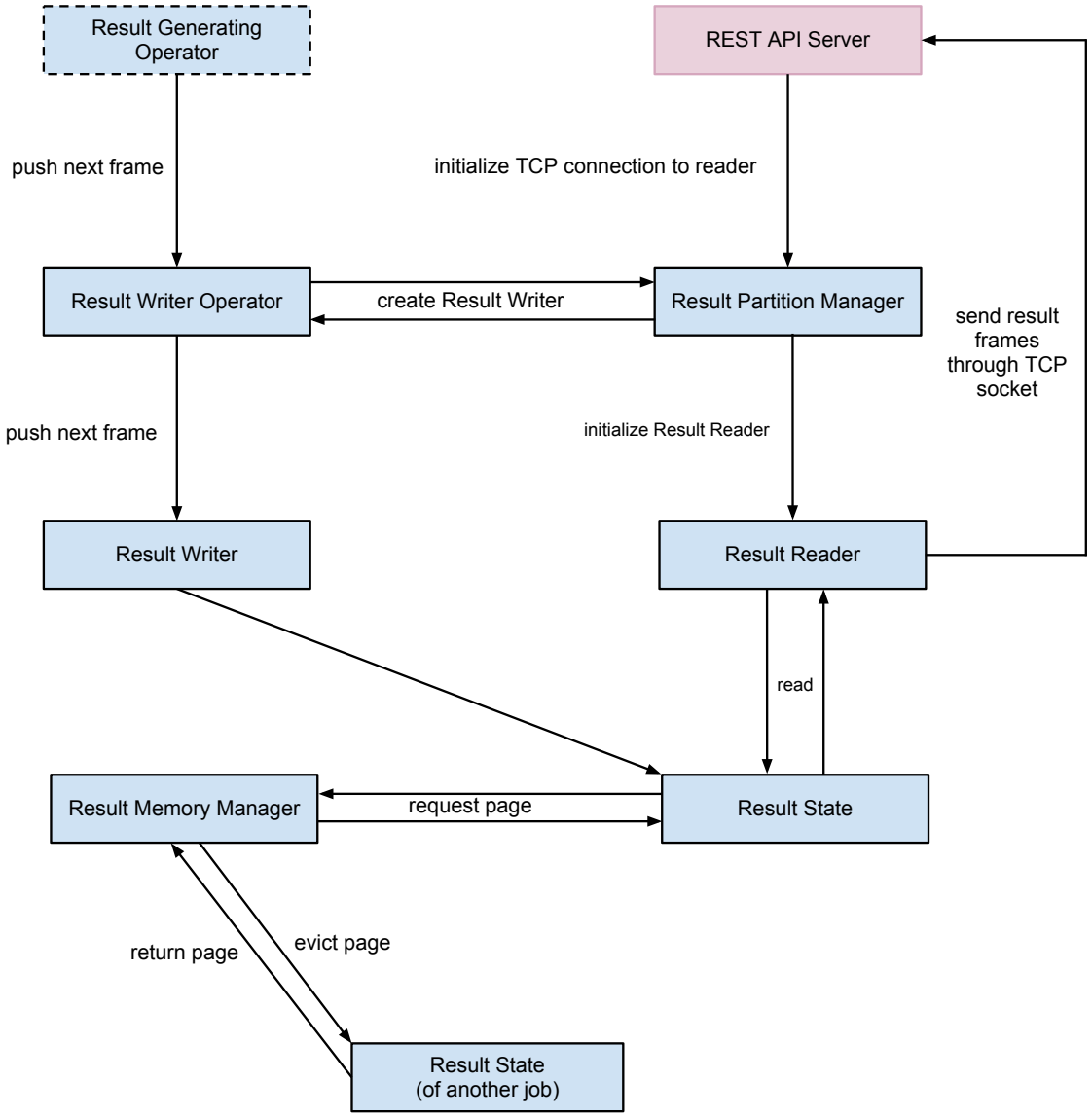


Figure 3.3: Result Distribution Architecture: View From A NodeController

responsible for overall result management at each Node Controller and Partition Memory Manager is responsible for managing result distribution buffer memory at each Node Controller. Hyracks, the runtime system also consists of a global memory manager that manages the memory for job execution and individual operators. But Partition Memory Manager is independent of this global memory manager and exclusively manages the result distribution buffer memory, while the global memory manager manages the rest of the Node Controller's memory. Result Writer operator transforms the result records to the required format and

gives them to the Partition Result Writer. Partition Result Writer coordinates with the Result Partition Manager and the Partition Memory Manager to store the results in memory or on disk. Result Partition Manager keeps track of where the result records for each result partition are stored in Result State instances. And the Partition Result Reader delivers the result records to the Result Client.

Let us now discuss in detail how these components interact with each other to deliver results to the Result Client. As the Result Writer operator receives the result frames from the previous operator, it serializes each record in the result frame to human-readable ADM string format and stores them back as a UTF-8 encoded byte-array in the result frame. After it accumulates a frame full of serialized result records, it hands over the frame to the Partition Result Writer. The Partition Result Writer then asks the Partition Memory Manager on the Node Controller for a block of memory from result distribution memory budget to store this result frame. Partition Memory Manager maintains a Least Recently Active (LRA) list of all of the jobs that have been allocated result memory on that Node Controller. It first checks if there is a freely available memory block to be allocated to serve the Partition Result Writer's request and, if so, allocates it to the Partition Result Writer. Otherwise, it will have to choose a victim job from the LRA list and select a memory frame that was allocated to the job earlier for possible eviction. The victim job's Result State gets to choose which frame in the list of frames that it holds should be evicted.

Result State tracks the list of memory frames in which a job's result records are stored as an ordered list. It chooses the frame at the front of this list, flushes its content to disk, and then returns that frame to the Partition Memory Manager. The Partition Memory Manager then returns this frame to the Partition Result Writer that requested for a new frame. Each time the Partition Memory Manager returns a frame to the Partition Result Writer, it updates the LRA list. The Result Writer then stores the frame containing result records in this returned frame. This process continues until the Result Writer operator has serialized all

the result records and all the result records have been stored either in memory frame or on disk by the Partition Result Writer.

Meanwhile on the Directory Service side, as soon as the Result Partition Manager registers the job partition's result location, it returns this location to the Result Client waiting for these locations (12). At this point, the Result Client can use one of the two policies to read the result partitions from the locations returned by the Directory Service depending on whether the query results should be ordered or not. In this chapter, we discuss the policy for the case where query results should be ordered and discuss an alternative policy in the next chapter when we discuss different possible policies. The Result Client sets up a duplex TCP connection to the Result Partition Manager running at the location returned by the Directory Service (13). The Result Client supplies the JobId and the PartitionId as part of result connection initialization, thereby expressing its interest in reading the result partition represented by the supplied JobId and the PartitionId. As soon as the result connection is established, the Result Partition Manager creates an instance of Partition Result Reader for the given JobId and PartitionId. This Partition Result Reader then starts sending result frames to the Result Client through the connection established by the Result Client (13). The Result Client then reads a single result frame sent by the Partition Result Reader from the TCP socket and returns it to the AsterixDB API Server's Query Handler (14) that called the Result Client Library in step (7).

The Query Handler calls the Result Client in a loop to read one result frame at a time. Each time the Query Handler calls the Result Client, it reads a frame sent by the Partition Result Reader from the TCP socket and this continues until all the frames of a given partition has been read. When the Query Handler makes the next call to the Result Client to read the another frame, Result Client knowing that it has already reached the end of stream for the current partition, sends a request to the Directory Service again to obtain the location of the next result partition (8). This time, it includes the list of result partition locations that

it knows to the request. The Directory Service compares this list of result locations that the Result Client already knows about with its own entry for the job and, if there are any new locations registered, it returns those locations to the Result Client (12). The Result Client then sets up a TCP connection to the Result Partition Manager at the next result partition location, expressing its interest in reading the result partition with the given JobId and PartitionId. This Result Partition Manager creates an instance of Partition Result Reader which starts sending the result frames for the given partition (13). The Result Client then starts reading the frames for this partition and returns them to the the Query Handler every time Query Handler calls the Result Client (14). This cycle continues until all the result partitions for the given query have been been read. Once the end of stream for all the result partitions have been reached, the Result Client signals the end of stream to the Query Handler which then stops calling the Result Client for the next result frame.

The Query Handler constructs a JSON array of ADM records for each frame that it receives in the loop and sends that array of records back to the client that is waiting for the query response (15). This way, the result for a query is delivered to the client in chunks, where each chunk is a UTF-8 encoded byte-array containing a single JSON array of ADM records. Note that, although the result is returned to the client in chunks, it all happens in a single request-response cycle. Thus, in the case of synchronous queries, the entire result set corresponding to a query, however small or large it is, is streamed back to the client in a single request-response cycle.

3.2.2 Asynchronous Queries

The Synchronous mode works well for queries that take a short time to run. If queries take too long to run, however, a client will have to keep its connection to the result distribution server open for a long period and, in turn, the Result Client will need to keep its connections

to the Result Partition Managers open until all the result partitions have been read. Keeping connections open requires resources and hence may impact the system's overall performance. Another disadvantage of the synchronous model is that, since a query's result is served in a single request-response cycle, if the client loses its connection to the server mid-way, there is no way to get the result for the query back. The query would then need to be submitted again and AsterixDB would deem the query as a new query and execute it again. Especially if the query is a long running query, this obviously impacts the system's performance. In order to mitigate these problems, we added support for an asynchronous mode for queries. The approach to result distribution for asynchronous queries is summarized in Figure 3.4.

The architecture of result distribution for asynchronous queries is similar to the architecture for synchronous queries. Hence, we will discuss only the differences between the two architectures (indicated by * next to the step number in Figure 3.4). As in the case of synchronous queries, the client first submits its query to the AsterixDB API server, but in this case a query mode parameter is set to asynchronous (1). The Query Handler extracts the query statement from the request and calls the AQLParser with the extracted statement as an argument which returns the syntax tree (2). The Query Handler then calls Algebricks with this syntax tree is as a call argument. Algebricks uses the syntax tree to compile the query, performs query optimizations and returns the Hyracks job description (3). The Query Handler submits the Hyracks job to the Cluster Controller, which returns the JobId (4) and the Query Handler sends the start job command to the Hyracks Cluster Controller (5). At this point, the Query Handler, instead of waiting for the result, constructs a *handle* for the query that serves to uniquely identifies the query throughout the life of an AsterixDB instance. The constructed handle is then returned as a response to the client (6*) rather than the query results themselves. The client can remember the query handle however it wants for later use.

AsterixDB now continues to execute the query. The query executes exactly the same way

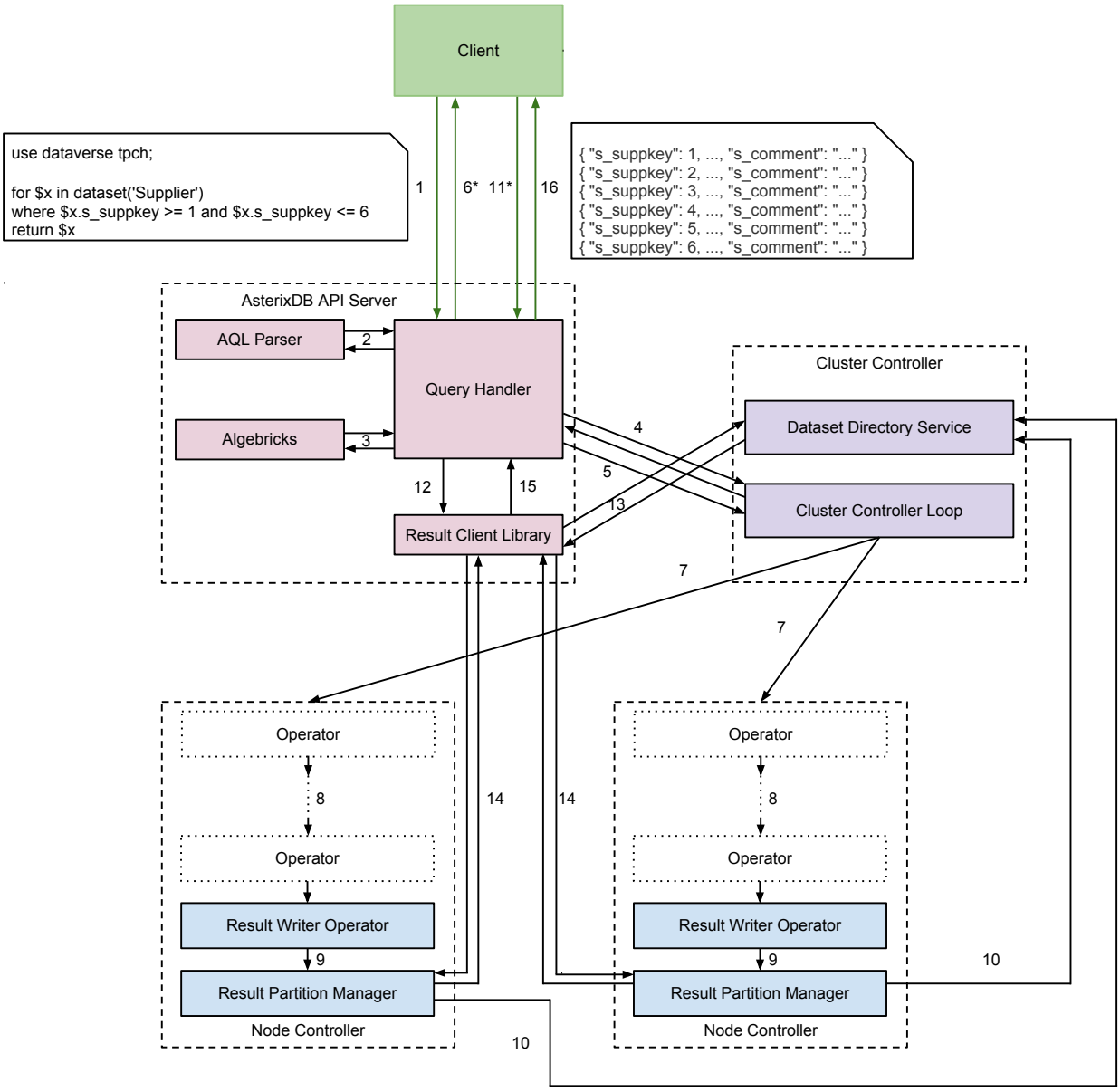


Figure 3.4: Result Distribution Architecture: Asynchronous Queries

as in the case of synchronous queries, all the way until the result reaches Partition Result Writer, which again stores the results in memory and/or on disk (7-10).

The query client can use the handle that it obtained at any time to request the result for the query. To do so, the client sends a query result request to the AsterixDB API server along with the query handle (11*). From there on, AsterixDB follows exactly the same sequence of steps to fetch the result as in the case of synchronous queries: The API server's

Query Handler invokes the Result Client (12) which asks the Cluster Controller to return the location of the Directory Service. The Result Client then connects to the Directory Service and requests the result partition locations (13). It then connects to each of the Result Partition Managers to read the result partitions (14). The Result Partition Managers intialize Partition Result Readers which then send the result records over the connection established by the Result Client. The Result Client in turn returns the result to the Query Handler (15). Finally, the Query Handler returns the result chunks containing JSON array of ADM records as a response to the query result request (16).

The major difference in the case of asynchronous queries is that the query and its results are served across two separate request-reponse cycles. Also, note that the client gets to decide when it should start reading the result for the query instead of keeping its connection to the AsterixDB API server open and waiting until AsterixDB executes the query and returns the result. This is useful for long-running queries because a client can submit the query, continue performing other functions without the need to keep a connection to AsterixDB open, and can then read the result later when it is ready to consume it. A disadvantage of asynchronous queries, of course, is that if the queries take very little time to execute and generate the result, two request-response cycles instead of one may add unnecessary overhead.

3.3 Lifecycle of Query Results

Result Partition Manager at each Node Controller stores the result for each query that it receives in memory and/or on disk. And the Directory Service stores the Directory Records containing the result locations and status of the result partitions for the queries executed by the system. Directory Service consists of an in-memory datastructure to store the Directory Records and the results are stored both in memory and on disk. Since, neither of

these resources are infinite, neither the Directory Records nor the results can be stored in AsterixDB for all the queries that were ever executed by AsterixDB. We need to frequently free these resources by deleting the results and the Directory Records of the old queries to make room for new queries.

In our architecture, we accomplish this by limiting the sizes of the datastructures that keep track of the result information. In the case of Directory Service, we limit the number of entries that the Directory Service can hold to a fixed number. Each entry in the Directory Service corresponds to a query. Everytime a new query is submitted for execution and a new entry is to be added to the Directory Service, Directory Service checks if the number of entries it holds is equal to its assigned limit. If so, it deletes the oldest entry in its entries map and adds the new entry to the map. Otherwise, it simply adds the new entry to the map.

The Result Partition Manager uses a similar strategy. It maintains a map of all the query partitions for which it is storing the result at any point. We limit the number of entries it can hold to a fixed number. Everytime a new query partition's Result Writer operator registers itself with the Result Partition Manager, the Result Partition Manager checks if the number of result partitions it holds is equal to its assigned limit. If so, it deletes the oldest result partition in its result partitions map, deletes all the result records corresponding to that partition stored on disk and adds the new entry to the map. Otherwise, it simply adds the new entry to the map.

If the client requests for the result corresponding to a query whose result partitions or the Directory Service entry has already been deleted, an error message is returned to the client indicating that the client is requesting for an unknown query.

3.4 Client Interface: AsterixDB API

The client interface exposed by AsterixDB is a simple REST-like API which exposes a number of API call endpoints. A client can simply encode its query as a request parameter and submit the request to the relevant API endpoint. The AsterixDB client interface provides five such endpoints:

1. Query: This is the endpoint to which AQL queries must be submitted. The query result mode, synchronous or asynchronous, can be specified as a request parameter.
2. Query Status: This is the endpoint to obtain the status of an asynchronous query.
3. Query Result: This is the endpoint to read the result of an asynchronous query.
4. DDL: This is the endpoint to submit DDL operations (e.g., create dataset). This endpoint does not return any result. If the DDL operation is successful, an OK response status code is returned, and otherwise a Server Error status code is returned.
5. Update: This is the endpoint to submit update operations (insert or delete). Like the DDL endpoint, this endpoint does not return any result, instead returning either OK or a Server Error status code.

Chapter 4

Implementation

In the previous chapter, we discussed the architecture of the AsterixDB result distribution framework. We also discussed the various components involved in result distribution and the roles that they play in aggregating, storing and distributing query results. In this chapter, let us look at the implementation details of the key components that were implemented as part of the initial AsterixDB result distribution framework. These components include: Directory Service, Result Client Library, AsterixDB API Server, Result Writer Operator and Result Partition Manager at Node Controllers. In addition to these components, we will also discuss the details of query plan generation to accommodate result distribution and various policies that were experimented with.

4.1 Directory Service

At the heart of the result distribution framework is the Directory Service. It is because of the Directory Service that the clients can find the results for their queries even after the queries have completed their execution. Figure 3.1 shows the location of the Directory Service in

AsterixDB. Queries are compiled into Hyracks jobs and are submitted to the Hyracks Cluster Controller for execution, which assigns a unique JobId for each job. Also, since Hyracks is a generic data-parallel runtime, a single Hyracks job can potentially generate multiple result sets. Each result set that such a job generates will be assigned a ResultSetId which is unique for the given job. In AsterixDB, however, a given query can generate only one set of results, so the query compiler generates only one ResultSetId for each job. The Directory Service is simply an in-memory, two-level map of JobIds, ResultSetIds, and their records that resides on the same cluster node(s) as the Hyracks Cluster Controller.

The Directory Service stores the JobIds of all the jobs as the keys of the top-level map. The value for each JobId is another map whose keys are ResultSetIds. This second-level map maps the ResultSetIds to an array of Directory Records. Each Directory Record in this array corresponds to a partition of a job's result. Hence, the length of this array is equal to the number of result partitions that the query generates, and the index of the array element represents the PartitionId of the partition represented by the Directory Record. Each Directory Record stores the network address and the status of the partition that it represents.

The Directory Service exposes an interface for result partitions to register their locations via the Result Partition Managers running on the corresponding Node Controllers. It also provides an interface for the result partitions to report their completion or failure. In addition to that, the Directory Service also exposes an interface for the Result Client Library to lookup a job's Directory Records or the job's status.

4.2 Result Client Library

The Result Client Library is a component of the result distribution framework through which the result partitions are fetched from the locations where they are stored and delivered to the client. Figure 3.1 shows the location of the Result Client Library in AsterixDB. The Result Client Library provides an interface called the "Result Reader" interface through which the AsterixDB API server reads the results for the queries. The Result Reader, as the name suggests, provides a reader-like interface. The AsterixDB API Server creates an instance of the Result Reader for each query with JobId and ResultSetId as its initialization parameters, and it starts reading the results through the read method provided by the interface. Each time the read method is called, the Result Client Library checks to see if it already knows about the Directory Records corresponding to all partitions of the given JobId. If not, it makes a Remote Procedure Call (RPC) to the Directory Service with the records it already knows about as the call argument. The Directory Service then compares the list of records that the Result Client Library already knows about with the list of records that the Directory Service knows about to check if there are any new records that the Result Client Library is not aware of. If there are new records, it returns those records to the Result Client Library. Otherwise, it blocks the call until new result partitions register their locations with the Directory Service. The Result Client Library makes an RPC to the Directory Service during each invokation of the read call, continuing until it obtains the Directory Records corresponding to all the partitions of a job.

The Result Client Library, upon receiving the list of Directory Records, extracts the network addresses from the records and sets up a TCP connection to each of the Result Partition Managers running at those locations. The Result Partition Managers then start sending the results they store for the given JobId as frames through their TCP connection. The Result Client Library reads each of these frames from the TCP socket and returns them to the AsterixDB API Server.

4.3 AsterixDB API Server

Figure 3.1 shows the components of the AsterixDB API Server. The AsterixDB API server is implemented as an HTTP server that listens to client API requests from multiple API endpoints. The implementation consists of an HTTP handler for each endpoint of the REST-like API. Each handler performs a specific function depending on the endpoint that it serves. The DDL, Update, and Query handlers serve the DDL, Update, and Query API endpoints respectively, by calling the AQL Parser to parse the AQL statements submitted by a client to obtain a syntax tree, and then calling the Algebricks algebraic compiler to compile the syntax tree into a Hyracks job and submitting the job to the Hyracks Cluster Controller for execution. The DDL and Update handlers just wait for the job execution to complete and report whether the job successfully completed or failed to the clients. The Query handler instead takes a different path. If the query is an asynchronous query, the Query handler constructs a "handle" for the query and returns it to the client. However, if the query is a synchronous query, the Query handler calls the Result Client Library to immediately fetch the result for the query. The Query handler calls the Result Client Library in a loop, fetching one frame of result at a time, extracting the result records from the frame and constructing a JSON array of result records out of it and sending it out as response to the client. Thus, the client receives the result records as chunks, where each chunk is a JSON array of ADM records.

For asynchronous queries, the QueryResult handler takes the query handle as a request parameter and calls the Result Client Library to fetch the result for the query represented by the handle. It fetches and returns the results to clients in exactly the same way as the Query handler fetches and returns the results to clients. The QueryStatus handler takes the query handle as a request parameter and calls the Result Client Library to obtain the status of the query represented by the handle.

4.4 Result Writer Operator

Figure 3.2 shows the role of the Result Writer operator in AsterixDB's result distribution framework. The Result Writer operator is another important component of the result distribution framework. A Result Writer operator is introduced into the query plan as the last operator of every AQL query. The operator just before the Result Writer operator in the query plan generates the result for the query in the binary format at each of the Node Controllers where it is scheduled to run. Then each of those Node Controllers initializes an instance of the Result Writer operator. As part of the initialization process, the Result Writer operator registers itself with the Result Partition Manager running on the local Node Controller. The Result Partition Manager creates an instance of a Result Partition Writer and returns it to the Result Writer operator. After the initialization, each frame from the previous operator containing result records in binary format is pushed to Result Writer operator; the Result Writer operator extracts the records in the frames, serializes them into human-readable ADM strings, and inserts the strings into result frames. These frames, containing serialized records, are then handed over to the Partition Result Writer for storage. The Partition Result Writer copies the result records in each frame that it receives and returns the original frame back so that the Result Writer operator can recycle the same frame for the next frame of result records. After handing over all the result records corresponding to a partition to the Partition Result Writer, the Result Writer operator reports to the Partition Result Writer that its task is complete, which is in turn reported to the Result Partition Manager.

4.5 Result Partition Manager

Figure 3.3 shows the role of the Result Partition Manager in AsterixDB's result distribution framework and the components with which it interacts. From the previous chapter, we already know that Result Partition Manager is responsible for overall result management at each Node Controller. When the Result Writer operator registers itself with the Result Partition Manager as part of the operator's initialization process, the Result Partition Manager makes an RPC to the Directory Service to register the partition's location, JobId, ResultSetId and PartitionId with the Directory Service. It also creates an instance of the Partition Result Writer and returns it to the Result Writer operator. When the Partition Result Writer reports task completion, the Result Partition Manager makes another RPC to the Directory Service to update the partition's status in the Directory Record.

The Partition Result Writer created by the Result Partition Manager is responsible for storing job results. An instance of Result State created by the Result Partition Manager is given to the Partition Result Writer for the purpose of storing a result partition's state. When the Result Writer operator hands over the result frame, the Partition Result Writer asks the Partition Memory Manager in the Result Partition Manager to allocate a block of memory to store the result frame. Upon receiving this frame, the Partition Result Writer copies the contents of the result frame that was handed over by the Result Writer operator to this new memory frame and makes a record of the new memory frame in the Result State. The frame in which the result was handed over to the Partition Result Writer is then returned back to the Result Writer operator so that it can recycle the frame for the next set of result records.

An important sub-component of the Result Partition Manager is the Partition Memory Manager; this is a resource manager that keeps track of the memory budgeted for result distribution buffering. This component keeps track of the free memory that is available

for storing result partitions and also the memory used by each job for storing its result partitions. It also maintains a LinkedList+HashMap of Least Recently Active jobs to track the activeness of jobs. When a Partition Result Writer requests a frame of memory to store the result records of a job, Partition Memory Manager checks if there are any free memory frames available. If one is available, the Partition Memory Manager allocates that memory frame to the job and updates the Least Recently Active datastructure. If there is no free memory available, the least recently active job is chosen from the datastructure as a victim and a memory frame that was previously allocated for that victim job is evicted. The victim job gets to choose which particular memory frame to evict. The Result State for that job chooses a frame to be returned to the Partition Memory Manager, flushes its content to disk, updates its records to indicate that the result records corresponding to that particular frame are now on disk instead of in memory, and then returns that memory frame. The contract here is that, if the Result State holds one or more memory frames, it has an obligation to return a frame to the Partition Memory Manager if asked. Were the Result State to violate this contract, the Partition Memory Manager would simply evict all of the memory frames that have been allocated to the victim job and add them to the free memory list without giving the Result State a chance to flush its content to the disk. (This should never happen in the current implementation.)

The result partitions stored by the Result Partition Manager are read by the Result Client Library. When the Result Client Library connects to the Result Partition Manager to fetch the result corresponding to a partition, the Result Partition Manager creates an instance of the Partition Result Reader to deliver the requested result partition to the Result Client Library. The Partition Result Reader reads the frames of result records sequentially and returns them to the Result Client Library in order. It uses the Result State to determine whether each frame that must be read is in memory or on disk and then reads the frame accordingly from that location and sends it to the Result Client Library via the TCP connection setup by the Library.

4.6 Plan Generation for Result Distribution

In order to deliver the results to the clients through the result distribution framework, the Result Writer operator should be the last operator of every query plan. It is the responsibility of the query compiler to introduce this operator into the query plan. To accomplish this task, AsterixDB introduces a logical operator called the DistributeResultOperator into the logical query plan as the last operator of the plan. Logical operators define a property called the partitioning property which defines the number of the input partitions of data they expect. This provides a policy opportunity: We can either specify the partitioning property as unpartitioned or we can specify that we want the cardinality of the input domain of this operator to be same as that of the previous operator. If the partitioning property is specified as unpartitioned, the entire result set of a query will be aggregated at one node. If we choose the cardinality to be the same as previous operator, then all the partitions generated by the previous operator will be handed over to the result writer operator instances on the same Node Controllers without any aggregation. We can choose either of these two policies. We will experiment with their implications in Chapter 5.

After the DistributeResultOperator is introduced into the logical plan, Algebricks optimizes the logical plan to generate an optimized logical plan. Since there are no optimizations that can be applied for result distribution, the DistributeResultOperator is simply retained, as is, in the optimized logical plan. Finally, while generating the physical plan for the optimized logical plan, Algebricks obtains an implementation of the serializer required to serialize binary data into the required result format from AsterixDB. Algebricks then creates an instance of the Result Writer operator, with the serializer, ResultSetId, and a boolean parameter specifying whether the query result partitions should be ordered or not as initialization parameters to the Result Writer operator. We will also explore the impact of this parameter in Chapter 5.

This Result Writer operator is then introduced into the physical plan along with a connector that defines how the data should be exchanged between Result Writer's previous operator and the Result Writer operator. If the partitioning property in the logical DistributeResultOperator is defined as unpartitioned, a random-merge-exchange connector is introduced, and otherwise a one-to-one-exchange connector is introduced into the physical plan.

4.7 Result Distribution Policies

There are various policies that one can use in result distribution. Here we discuss some of the basic policies that we have experimented with in our implementation of result distribution in AsterixDB.

4.7.1 Centralized vs Distributed Result Distribution

The partitioning property in the logical operator for result distribution defines the number of result partitions that should be generated for the query. This provides an opportunity for two different policies.

As we have already discussed in the plan generation section, we can define a compilation rule to aggregate the result of a query into one partition before delivering it to the client. An advantage of this approach is that the results will be aggregated at a Node Controller, which is a worker node in the system, instead of aggregating the result at the Result Client Library which lives in the AsterixDB API server as shown in Figure 3.1. A disadvantage of this approach is that, since all of the results for a query are aggregated at one node, and should be stored before they are read by the client, it leads to higher disk pressure (reads and writes to the disk) on the disks that store the results for large queries.

An alternative policy is to generate as many result partitions as the query's operator just before the Result Writer operator generates. An advantage of this policy is that, since each query's result is distributed across multiple nodes, the result disk pressure (writes and reads from disk) is also distributed across multiple nodes and hence across multiple disks.

4.7.2 Sequential vs Opportunistic Result Distribution

When the query specifies an ordering, the result records should be delivered in the same order as they were generated during query execution. However, when the query does not specify any ordering, it provides an opportunity for different policies. The result records can either be delivered in the same order they were generated or in any arbritrary order.

One of the possible policies is that, for an unordered query containing N result partitions, the Result Client Library can read the partitions sequentially, starting from partition 0, partition 1, to partition N-1, one after the other. A disadvantage with this policy is that the result reading rate is bounded by the partition that has the slowest result production rate, and the production process must wait for the each result partition to be available in order.

Instead of reading the result partitions sequentially, the Result Client Library can opportunistically read the result records from all the partitions simultaneously. In this policy, the Result Client Library connects to all the Result Partition Managers containing the result partitions for a query simultaneously and reads the result records from them as they become available. There is no order at all here. An advantage of this policy is that the Result Client Library will not be blocked by a result partition with a slow production rate. It instead continues reading the result records from all the partitions as they appear.

4.7.3 With vs Without a Result Distribution Buffer

The Result Writer Operator hands over the result frames generated for the queries to the Result Partition Manager, which then decides where to store the result frames, i.e., either in memory or on disk, until they are consumed by the client. Memory is a critical resource since it is required for all types of operations in the system. Hence, memory should be budgeted for each task carefully. Therefore, there are two possible policies here.

One policy is to allocate jobs a certain memory budget for use as a result distribution buffer. We can use this memory to store the result frames. Since AsterixDB is a Big Data system designed for queries with potentially large result sets and high concurrency workloads, not all the result frames belonging to all the queries can be stored in memory and some memory frames that were allocated to the jobs earlier should be evicted to store the result frames belonging to new jobs. This opens up an opportunity for sub-policies to decide which frames should be stored in memory and which frames should be stored on disk. The current implementation uses a list of Least Recently Active (LRA) jobs, i.e., an ordered list of jobs where the jobs whose result frames were recently read or written are at the back of the list. In other words, the job at the front of this LRA list is the least recently active job. This least recently active job at the front of the list is chosen as a victim job and one of the result frame that was allocated to it earlier is selected for possible eviction. The job chooses a frame for eviction, flushes its content to disk and returns the evicted frame so that it can be used to store the result records of another job. We will not explore the details of alternative possible sub-policies in this thesis.

An advantage of using a result distribution buffer is that, if the query results can be held in memory until they are read by the client, we can avoid the cost of two disk I/Os: one to write the result frame to disk and another to read it back from the disk to deliver it to the client. However, the disadvantage is that memory that could have been used for some other

operation in the system is now used for result distribution buffering.

An alternative policy is to not allocate any memory for result distribution buffering at all. All the result frames are always written to disk. The hope in this case is that, if the nodes are not under heavy memory pressure, and if there is enough room for the filesystem to cache the disk reads and writes, the filesystem cache will still aid in improving the performance of disk I/Os. The advantage in this case is the simplicity of the implementation. Result frames do not have to go through memory allocation, flushing, and all the related buffering trickery within the Result Partition Manager. However, when the system is under heavy load, the filesystem cache may not come to the rescue.

Chapter 5

Experiments

A goal when designing and implementing the result distribution framework for AsterixDB was to provide better result management than in traditional databases or than what Hadoop-style Big Data systems [6] offer. The result distribution framework that we have implemented, unlike those of relational databases and interfaces like JDBC [7], decouples result production from consumption. As a result, client result consumption rates do not throttle AsterixDB's result production rates. AsterixDB's runtime produces results for queries independent of consumption, and the result distribution framework takes care of temporarily storing the results and delivering them to clients once they are ready to consume them. Big Data systems such as Hadoop write their results to files and expect the clients to read the results from those files. The disadvantage of that approach is that clients cannot start reading the results for a query until the result set has been completely written to the file. In contrast, our result distribution framework allows clients to read results as soon as they begin to be available. Hence, the AsterixDB approach provides the advantages of the cursor-style result delivery techniques used in traditional databases along with the ability to temporarily store the results in memory or on disk for handling large result sets.

In this chapter we will present a set of experiments that explore the basic performance of the AsterixDB result management framework. Additionally, in the previous chapter, we discussed various policies that are possible with our result distribution framework. In this chapter, we also present results from experiments that we have conducted with different combinations of these policies to explore which policies are better in what circumstances.

5.1 Experimental Setup

Since our framework and its policies are aimed at clusters under pressure due to multi-user workloads, we use a small 5-node cluster setup for our experiments. One of the nodes of this cluster has an Intel(R) Core(TM)2 Duo E8500 processor, 4GB of RAM and a single 500GB disk. This node has a public IP address and is accessible from outside the subnet. We deploy the Hyracks Cluster Controller on this node and hence we refer to it as the Cluster Controller node (or CC for short). We also deploy the AsterixDB API server on this node. The remaining nodes of the cluster are identical; each has one AMD Athlon(tm) X2 3250e dual core processor, 4GB of RAM, and a 500GB hard disk. We run Hyracks Node Controllers (NCs) on these nodes. These nodes are accessible only from within the subnet and are only accessed through the Cluster Controller node. The cluster is interconnected via a 1Gbps ethernet.

AsterixDB and all of its components are implemented in the Java programming language and hence run within a JVM. We allocate 2GB of memory as the maximum Java heap size (-Xmx) for these JVMs, i.e. for the Cluster Controller and Node Controller JVMs. The unit of data communication throughout AsterixDB is a frame, and we set 32KB as the frame size. We allocate 128MB of memory for AsterixDB's buffer cache. For the experiments where we enable result distribution buffering, we separately allocate 128MB for the result buffer.

We drive the system through an in-house client workload generator tool called HERDER [1] that was built for benchmarking different types of Big Data systems. HERDER provides an API to implement clients for any type of system that provides an interface to submit jobs. We implemented an AsterixDB HTTP client using HERDER's interface, which is in-turn used by HERDER to submit queries to AsterixDB and read the query results. All of our time measurements are end-to-end and are as measured by the client, i.e., by HERDER.

5.1.1 Data and Queries

We used TPC-H's dbgen tool [8] to generate the data for the experiments. We generated 100GB of TPC-H data split into 4 chunks and then bulk loaded those 4 chunks in parallel into AsterixDB. Since we measure time end-to-end, and since the main focus of these experiments is to evaluate the result distribution framework's performance (and not that of the whole AsterixDB system), we want the actual query execution times to have as little impact on the time measurements as possible. We have thus chosen two simple primary-key range queries on the TPC-H Supplier table for our experiments. We call these two queries $\mathbf{Q_{small}}$ and $\mathbf{Q_{large}}$.

Query $\mathbf{Q_{small}}$ in AQL is as follows:

```
for $x in dataset Supplier
where $x.s_suppkey >= <start_range> and $x.s_suppkey < <start_range>+10000
return $x
```

The starting key of the range for this query $\mathbf{Q_{small}}$ is randomly chosen by HERDER from the list 1, 10001, 20001, ... 90001.

Query $\mathbf{Q_{large}}$ in AQL is as follows:

```
for $x in dataset Supplier
where $x.s_suppkey >= 1 and $x.s_suppkey <= 1000000
return $x
```

In order to avoid the impact of inter-query buffer caching on the experimental results, 20 replicas of the Supplier dataset are loaded into AsterixDB while bulk loading the data into the system. Each query is randomly submitted to one of the replicas. The probability of choosing a replica for a query is calculated by HERDER based on probability configuration parameters defined in HERDER's workload definition file. Herder provides a notion of a query stream, where each stream executes a class of queries. In our setup, we define one stream called "Small-Query", which consists of instances of query $\mathbf{Q_{small}}$, and another stream called "Big-Query", which consists of query $\mathbf{Q_{large}}$ instances. When we run the experiments, HERDER runs the streams in parallel. We can define the number of threads of execution, called the population count, within each stream in the HERDER workload definition file. For the experiments here we vary the population count from 1 to 4 per stream. An instance of an experiment thus involves running the two streams in parallel with the given population count per stream and the policy chosen for the experiment. We let the streams run for 900 seconds (15 minutes) in steady state for each instance of the experiment.

5.2 Evaluation of Various Policies

Let us begin our experiments by evaluating the performance of various combinations of result handling policies. To start with, we consider two different possible policies for each of the two different categories. This gives us four different combinations of policies. The two categories of policies we consider are: sequential vs opportunistic result reading and with vs without result distribution buffering.

As a reminder, for unordered queries, AsterixDB can read the result partitions sequentially, in the order in which they are generated, or opportunistically, as the result frames become available for each partition. Each policy has advantages and disadvantages as discussed in the previous chapter. In addition, for each of these two policies, we can either have result distribution buffering enabled or disabled. We will evaluate how the combinations of these policies perform. Figures 5.1a, 5.1b, 5.2a, and 5.2b (respectively) present the response times and throughputs for query $\mathbf{Q_{small}}$ under the four different policies. Figures 5.1c, 5.1d, 5.2c and 5.2d present the response times and throughputs for query $\mathbf{Q_{large}}$ for the same four policies. We analyze these results in detail below.

In all combinations of policies for both classes of query, the response time increases as the number of queries that run concurrently increases. This is because, as the concurrency level increases, system resources have to be shared among the queries; this leads to increased CPU, I/O and memory contention among concurrently executing queries, thereby increasing the response time of the individual queries. Their throughputs increase as we go from no concurrency to a concurrency level of 2 because, although the response time increases, the system can now exploit parallelism and hence can do more work, thereby increasing the throughput. However, as we increase the concurrency level further, resource contention increases, and after some point contention can increase to such an extent that it outweighs the benefits of parallelism and hence the throughput can start dropping. We can see this phenomenon in some of our plots in the following experiments. Let us begin with the first experiment.

5.2.1 Impact of Result Buffering (Sequential)

In this experiment, AsterixDB reads the query result partitions sequentially, one after the other, and we compare how the system performs for synchronous small result queries and

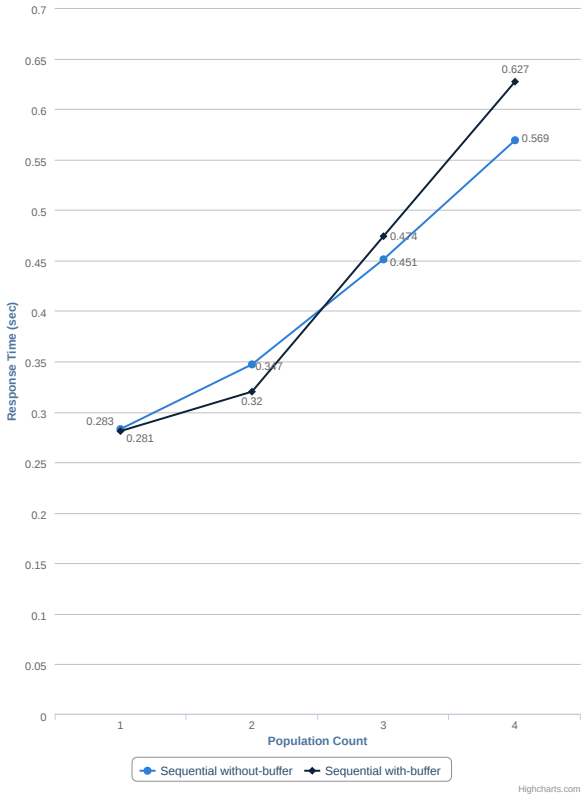
synchronous large result queries both with and without result distribution buffering.

From Figures 5.1a and 5.1b we can see that for smaller queries, buffering does not offer any improvement in the response time or throughput at lower concurrency levels, and as the concurrency increases, the throughput levels off and the response time increases. It seems likely that this is because, in our experiments small result queries run concurrently with large result queries, and since large result queries need more memory to store their results, smaller queries either have to write their results to disk or evict memory frames held by other queries. Evicting a page has a higher relative overhead compared to the response times of smaller queries because, the smaller query now has to wait until that memory frame is flushed to disk along with going through the mechanics of buffering to update all the records while evicting pages. Hence, the throughput is lower and the response time is higher for small result queries at higher concurrency levels when result distribution buffering is enabled.

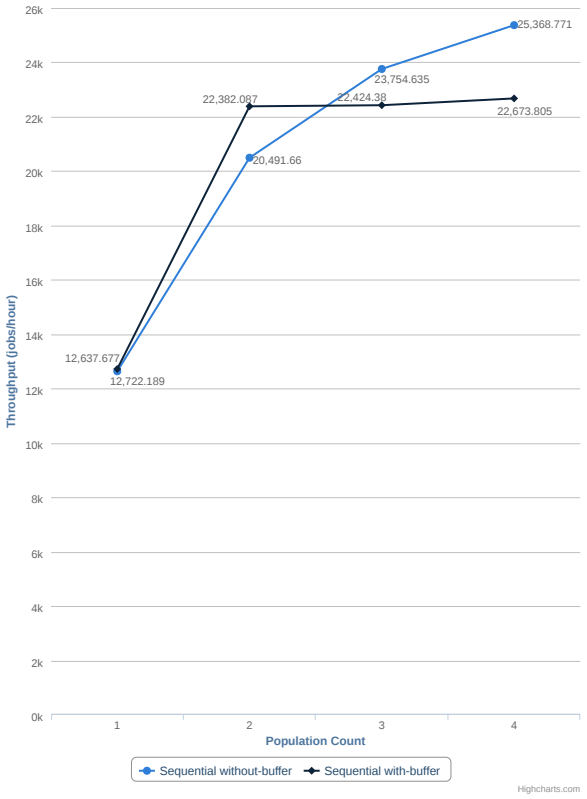
For large result queries, the performance characteristics are different. As we can see from Figures 5.1c and 5.1d, while buffering does not offer any improvement at lower concurrency levels, as the concurrency level increases, their response time and throughput both benefit. Since larger proportions of the results for large result queries are stored in memory (vs. on disk), and since memory accesses are orders of magnitude faster than disk accesses, the benefits of result distribution buffering are evident. Hence, the system provides better performance for large result queries when we buffer results.

5.2.2 Impact of Result Buffering (Opportunistic)

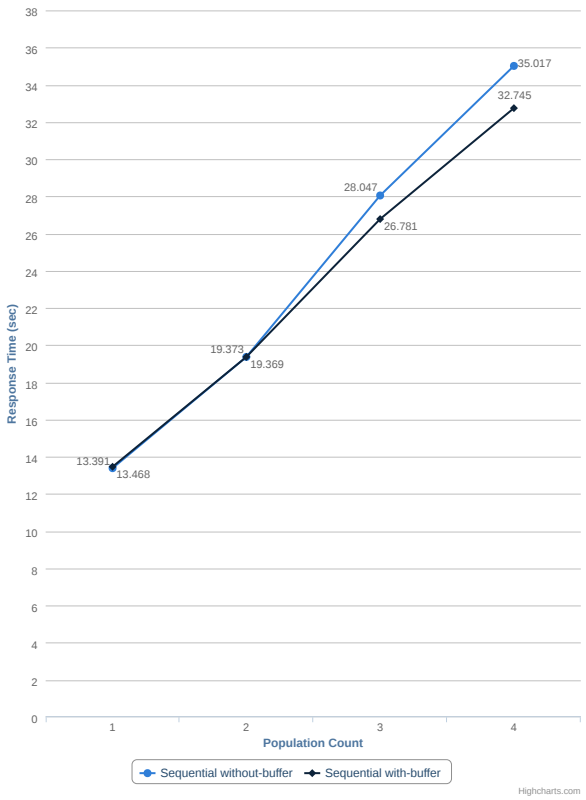
In this experiment, AsterixDB reads the result partitions opportunistically, i.e., as the result frames become available. We again compare how the system performs for small result queries and large result queries both with and without result distribution buffering.



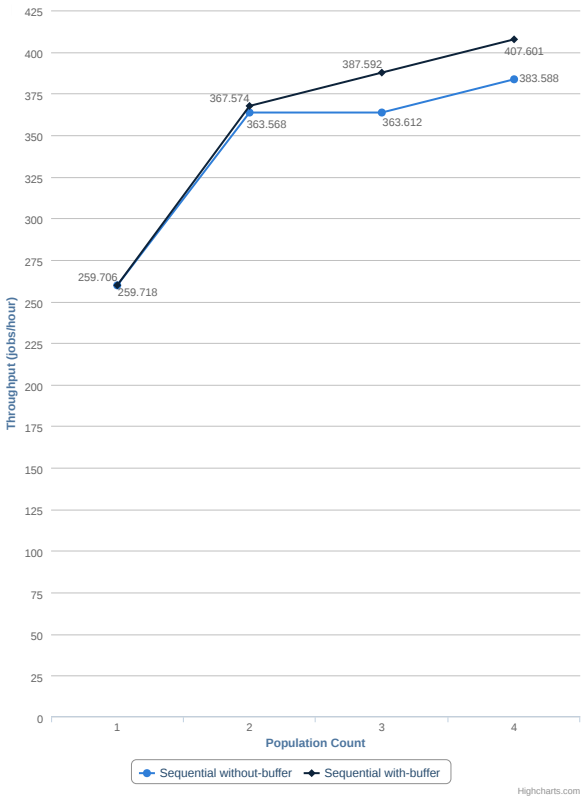
(a) Query $\mathbf{Q_{small}}$: Response Time



(b) Query $\mathbf{Q_{small}}$: Throughput

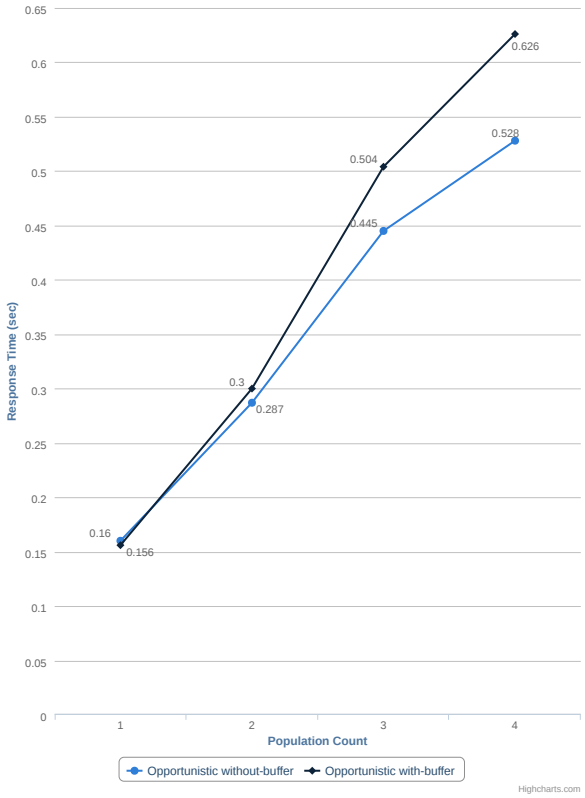


(c) Query $\mathbf{Q_{large}}$: Response Time

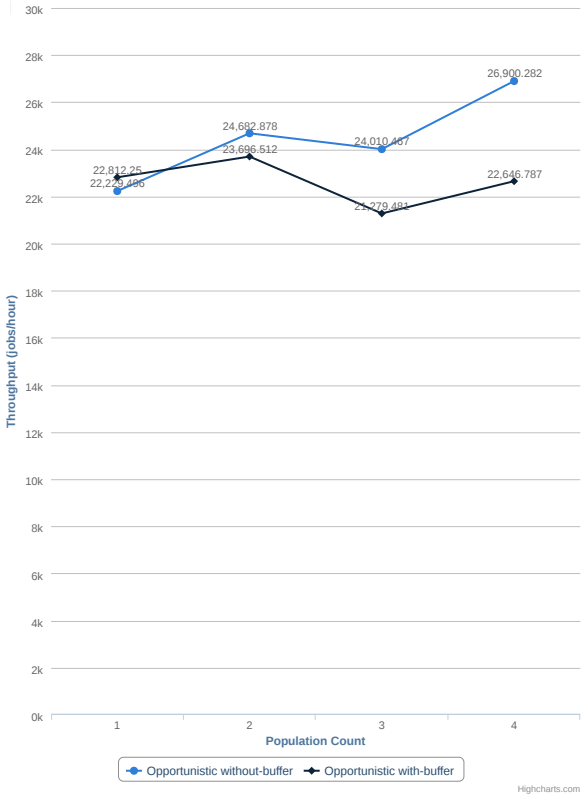


(d) Query $\mathbf{Q_{large}}$: Throughput

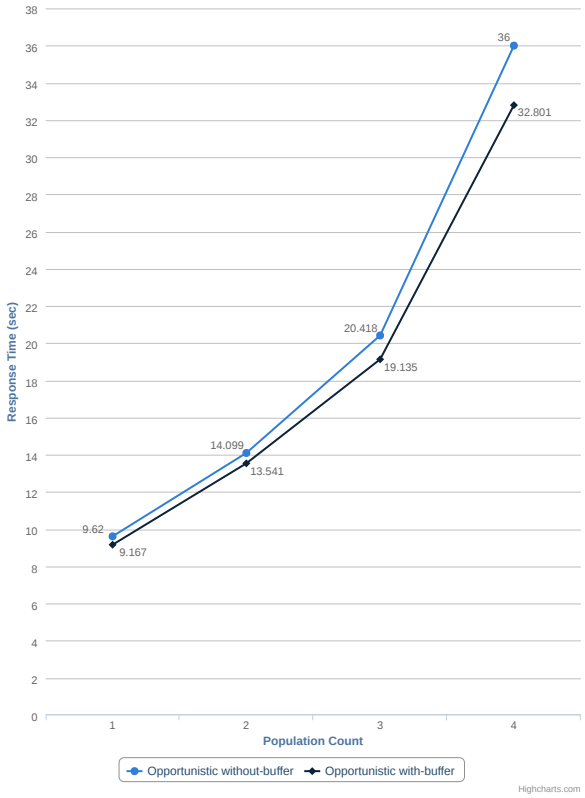
Figure 5.1: Sequential Result Reading - With Buffering vs Without Buffering



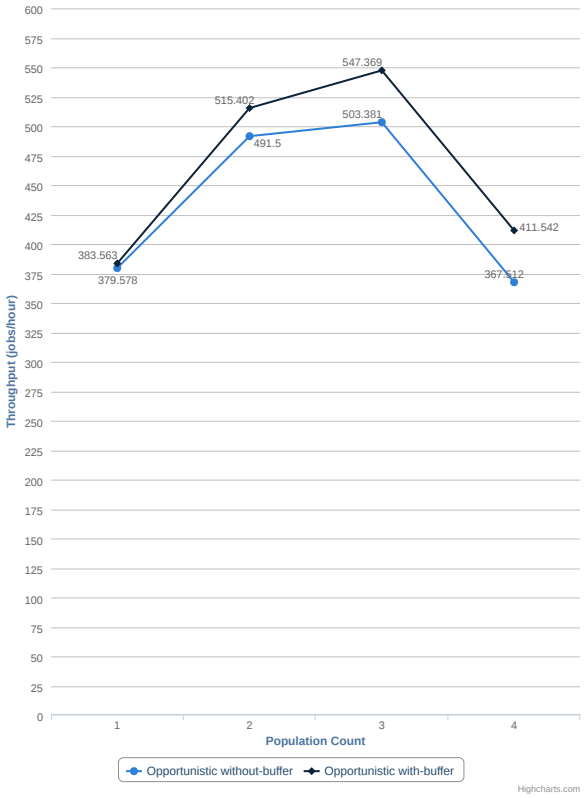
(a) Query $\mathbf{Q_{small}}$: Response Time



(b) Query $\mathbf{Q_{small}}$: Throughput



(c) Query $\mathbf{Q_{large}}$: Response Time



(d) Query $\mathbf{Q_{large}}$: Throughput

Figure 5.2: Opportunistic Result Reading - With Buffering vs Without Buffering

From Figures 5.2a and 5.2b, we can see that smaller result queries have similar performance characteristics as in the case of sequential result reading. That is, buffering does not offer any improvement for their response time or throughput at low concurrency, and as the concurrency increases, their throughput flattens out. The reasons are the same as those for the sequential result reading policy. Also, as seen in Figure 5.2b, the throughput in the cases of small queries with no concurrency and small queries with a concurrency level of two are about the same (for both the buffering and non-buffering cases). Since the system executes two small result query streams concurrently, each takes twice the time as a single stream of small result queries and the throughput remains the same. Similarly, since the system is essentially doing as much work as it can at the concurrency level of two, there are no significant differences in throughput at higher concurrency levels. (The throughput levels are roughly the same at all the higher concurrency levels, but the plot has variations, possibly because of the sensitivity of the small result queries to other processes that are active in the system.)

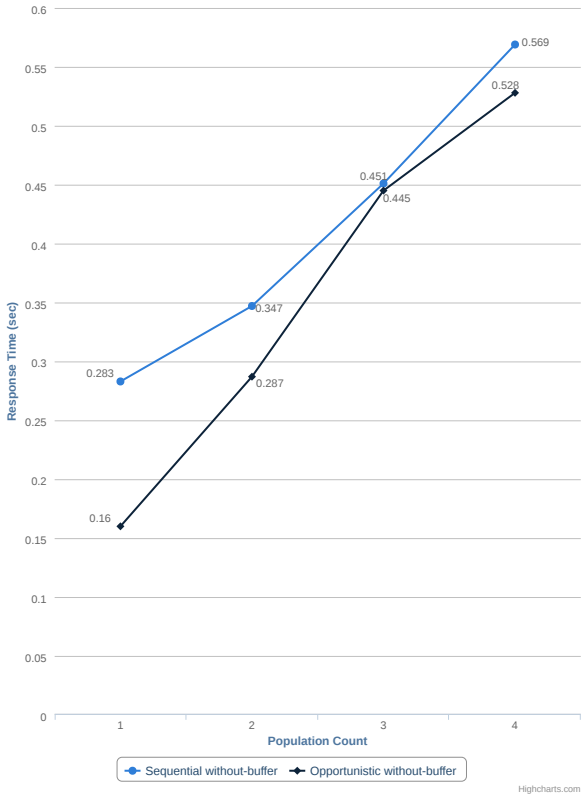
Turning to the large result queries, their performance characteristics are also initially similar to the sequential result reading policy's performance characteristics. As we can see in Figures 5.2c and 5.2d, buffering does not offer any improvement at lower concurrency levels, but as the concurrency level increases, the response time is lower and the throughput is higher when buffering is employed. Again, the reasons here are same as for the sequential result reading policy. In this experiment, however, the large query throughput drops in Figure 5.2d when we execute four large result set queries concurrently with four small result set queries. This is an interesting phenomenon that needs further investigation through additional experiments, and we leave this investigation as planned future work.

5.2.3 Impact of Sequential vs Opportunistic Result Reading

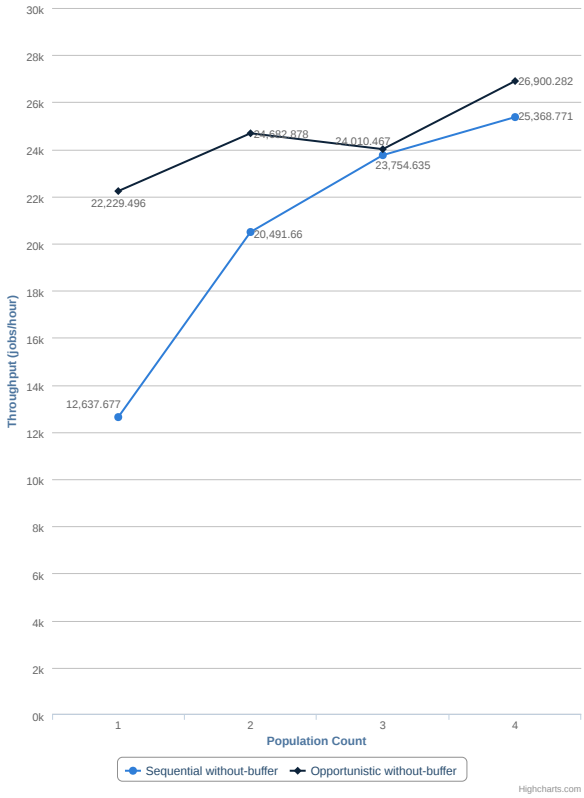
In this experiment, we compare the performance characteristics of reading result partitions sequentially vs. opportunistically for unordered queries. We only examine the case where result distribution buffering is disabled. (From the plots in the previous sections, it is not hard to extrapolate that the performance characteristics for the two policies that are being compared here would be similar when result distribution buffering is enabled.)

From Figures 5.3a - 5.3d we can see that opportunistic result reading performs better than sequential result reading at lower concurrency levels. At lower concurrency levels, the system has to aggregate and deliver results for fewer queries, and hence it may have to wait until result frames become available. However, when the result partitions are read sequentially, even when the frames of the partitions that are ahead of the current partition are available, the system does not read and deliver those partitions to the client until the current partition is completely read. Hence, the system may have to wait until the current partition is completely produced. This waiting may lead to lower resource utilization in the system. On the other hand, when we read the result frames opportunistically from all the partitions, the system can maximize its resource utilization and hence yield better performance.

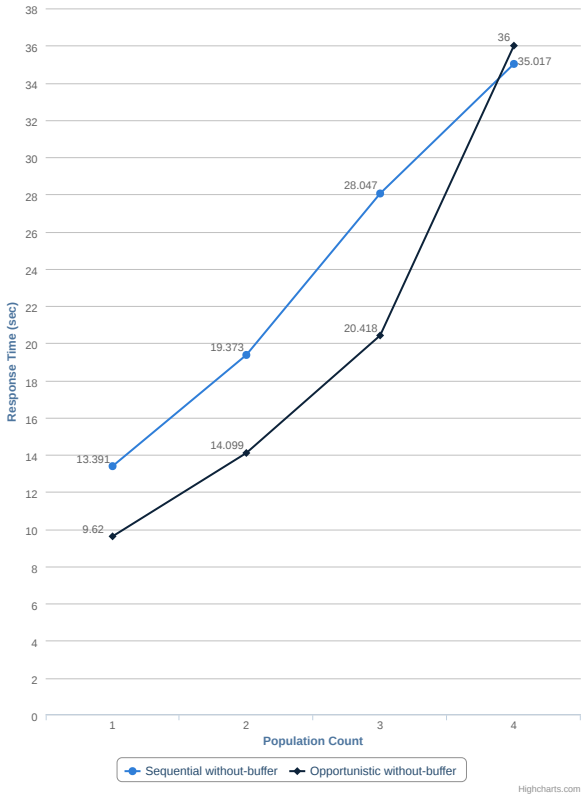
As the concurrency level increases, the difference between the performance of the two policies reduces, i.e. at very high concurrency levels, the response times and throughputs of the two policies become comparable (for both small and large queries) as seen in the figures. It seems likely that this is because, at higher concurrency levels, if the system (when reading the result partitions sequentially) is made to wait for a slow producing partition, the system's resources, which would otherwise be idling, can be used by other concurrently executing queries whose partitions are available to be read. This already maximizes the system's resource utilization, so the opportunistic result reading policy cannot improve the resource utilization anymore than the sequential result reading policy at higher concurrency levels. It thus performs



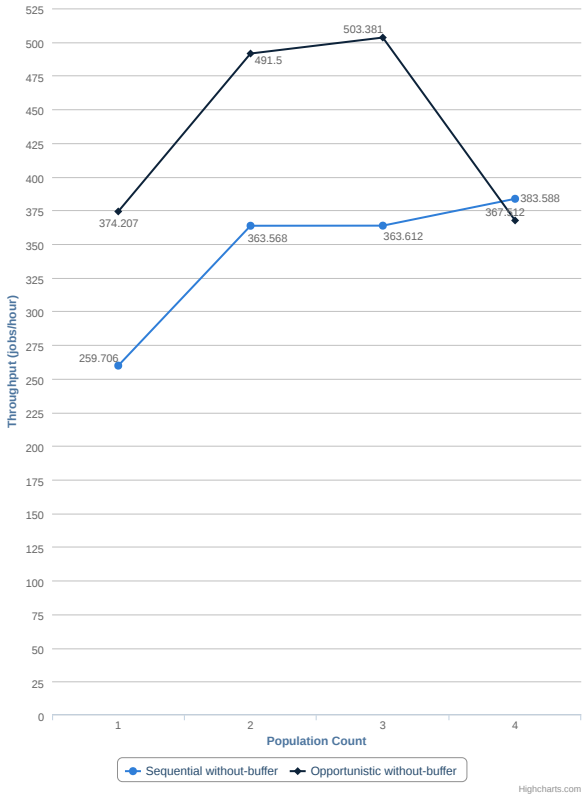
(a) Query $\mathbf{Q_{small}}$: Response Time



(b) Query $\mathbf{Q_{small}}$: Throughput



(c) Query $\mathbf{Q_{large}}$: Response Time



(d) Query $\mathbf{Q_{large}}$: Throughput

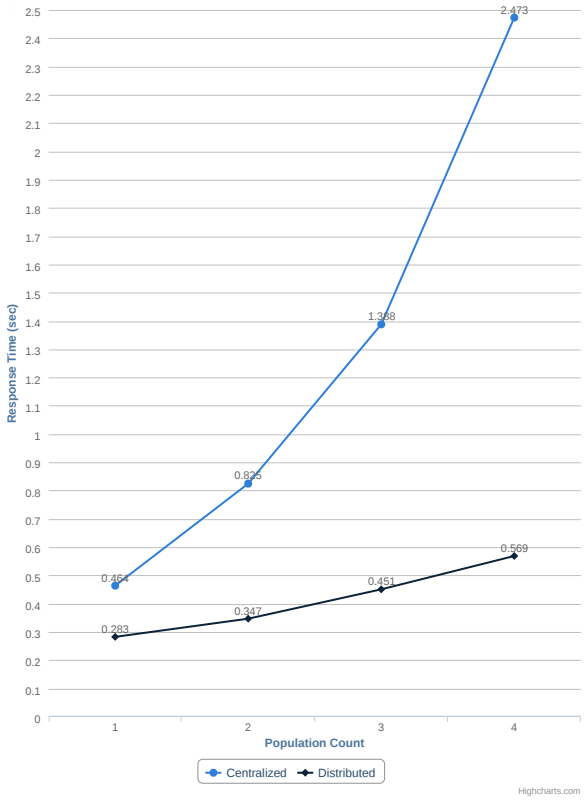
Figure 5.3: Sequential vs Opportunistic Result Reading

similar to the sequential result reading policy in this region of operation.

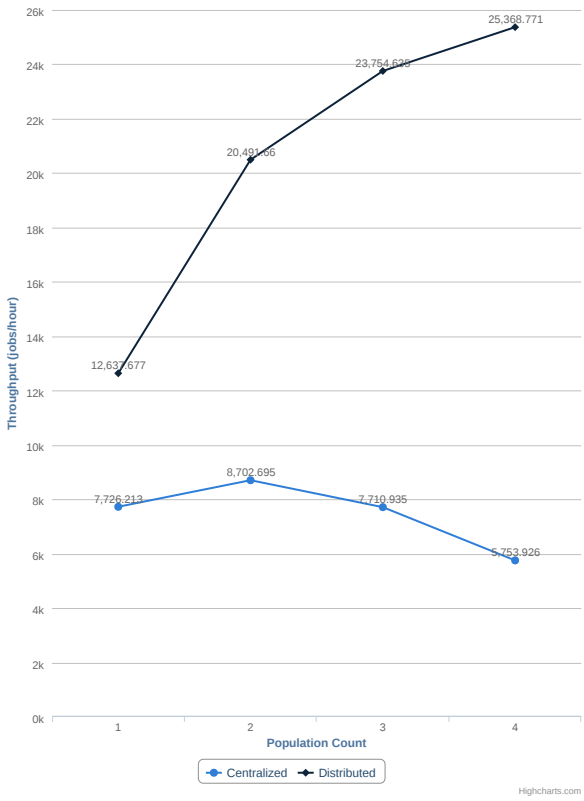
5.2.4 Distributed vs Centralized Result Distribution

In the case of centralized result distribution, the entire result set of a query is aggregated at one node before storing it and delivering the stored results to clients. Each query will have only one partition of results in this case. In the case of distributed result distribution, the result partitions for the queries are stored in the nodes where the last operator of the query before the Result Writer operator generated its results. In this experiment, we compare the performance of centralized result distribution with that of distributed result distribution. We use the sequential result reading policy and disable result distribution buffering for this experiment.

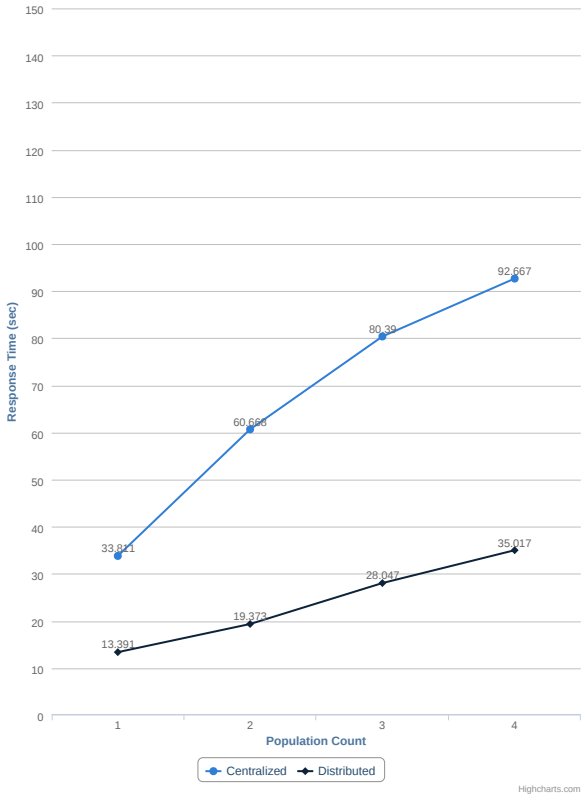
As we see in Figures 5.4a - 5.4d, distributed result distribution performs better than centralized result distribution at all concurrency levels. With distributed result distribution, both classes of queries have better response times and the overall system throughput is higher as well. We can also see that the difference in the performance between the two policies is always significant and increases as the concurrency levels increase. This is because, when the results are delivered to a single node for storage in the case of centralized result distribution, it increases the memory pressure and disk pressure on the node where the results are being aggregated and stored. In the case of distributed result distribution, the memory and the disk pressure are distributed across the nodes, hence providing faster response times and better throughput.



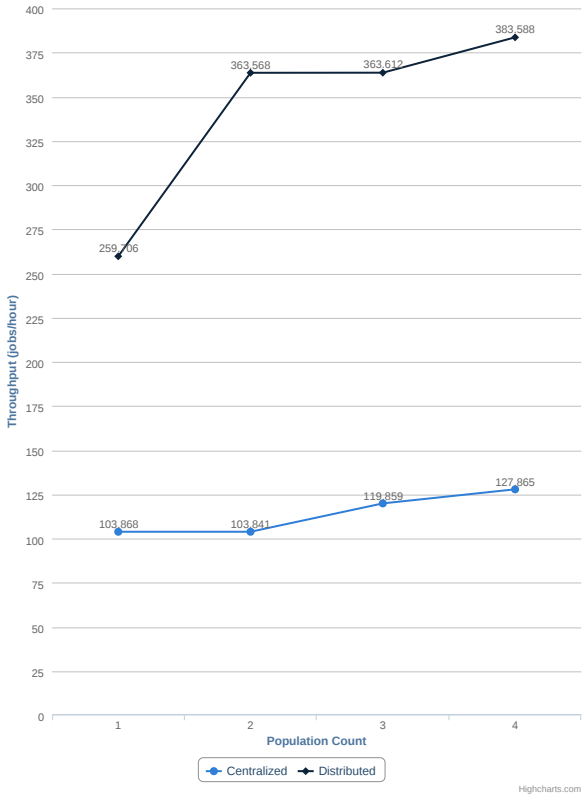
(a) Query $\mathbf{Q_{small}}$: Response Time



(b) Query $\mathbf{Q_{small}}$: Throughput



(c) Query $\mathbf{Q_{large}}$: Response Time



(d) Query $\mathbf{Q_{large}}$: Throughput

Figure 5.4: Centralized vs Distributed Result Distribution

5.3 Time To First Result

We discussed earlier that a major advantage of our implementation of the AsterixDB result distribution framework is that it can provide the advantages of a cursor-style result delivery technique while also allowing the system to manage large volumes of results that Big Data systems are expected to handle - doing both without throttling the system's performance. In order to demonstrate this advantage, in this experiment we compare the time-to-first-result, i.e. the time required to fetch the first result record for the query under two different scenarios.

The first scenario is the one that we have implemented and enabled by default in the result distribution framework in AsterixDB: We eagerly start reading the result records as they become available. The second scenario attempts to simulate a traditional Big Data result delivery mechanism where the results are written to files and then clients are expected to read the result from those files, i.e., where clients cannot start consuming the results until job execution is complete. The time is measured end-to-end, i.e. the time spent by the client waiting for the first result record to appear after sending the query through the API. For this experiment, we use the sequential result-reading policy with result distribution buffering turned off. We only consider query $\mathbf{Q_{large}}$'s response time.

As we can see from Figure 5.5, the time-to-first-result remains almost a constant even as the concurrency level increases when we eagerly read the results. This demonstrates the cursor-style advantages of our result distribution framework. On the other hand, if we use the technique of reading the result only after job completion, the time-to-first-result increases as the concurrency level increases, i.e., as the system's workload increases, the time-to-first result increases too. Clients often perform operations on the result records that they receive from their queries. If we can deliver the result records as early as possible, clients can start performing those operations on the records as they receive them, thereby overlapping

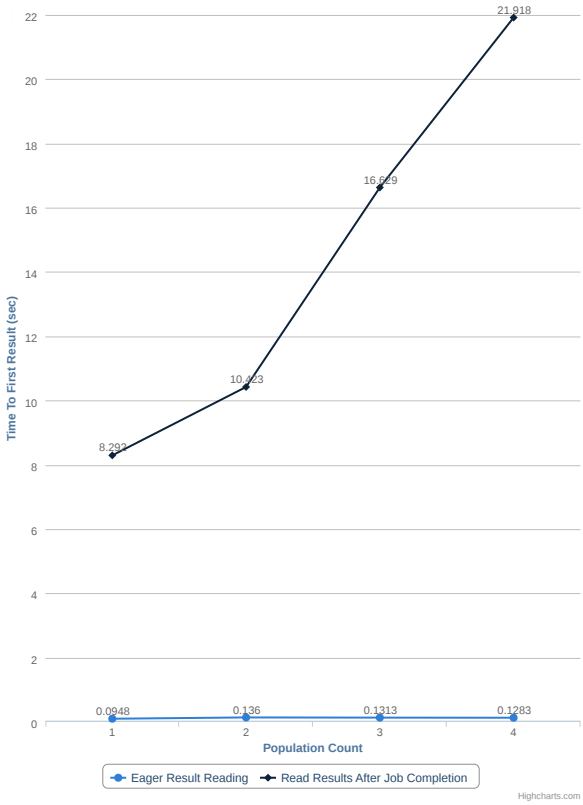


Figure 5.5: Time To First Result: Reading Results Eagerly vs After Job Completion

query execution with client side processing of records instead of waiting for job execution to complete for result records to appear.

Chapter 6

Conclusion

In this thesis we have presented the architecture of a result distribution framework that caters to the needs of result management in Big Data systems. We have also presented an implementation based on this architecture in AsterixDB. The AsterixDB result distribution framework consists of an AsterixDB API server, a Directory Service which lives within the Hyracks Cluster Controller, and Result Partition Managers at each Node Controller. The AsterixDB API server hosts the API endpoint query handlers, the AQLParser and the Algebricks algebraic query compiler, and the Result Client Library which coordinates the delivery of results from the locations where they are generated at Node Controllers to the clients through the query handlers. The Directory Service keeps track of the result locations for the queries executed by the system. The Result Partition Managers, along with their sub-components at each Node Controller, manage the query results at the location(s) where they are generated. The AsterixDB result distribution framework supports synchronous queries and asynchronous queries. For synchronous queries, AsterixDB eagerly delivers the results to the clients as a response to their queries. For asynchronous queries, query handles are returned to the clients as a response to their queries. The clients can store these handles and use them later to read the results whenever they want. In this thesis, we have also presented

several combinations of result distribution policies that are possible with the AsterixDB result distribution framework.

In addition to presenting various result distribution policies, we have also presented a set of experiments that explore the performance of the result distribution framework under various combinations of these policies. The experiments showed that the distributed result distribution mechanism performs better than centralized result distribution mechanism because it distributes the memory and disk pressure caused by result distribution across the worker nodes. The experiments also showed that our approach for result distribution provides the benefits of cursor-style result delivery techniques by delivering the result records as soon as they are available while still being able to handle the large result sets that Big Data systems are expected to handle. Further, the experiments showed that the opportunistic result-reading policy performs better than the sequential result-reading policy for large result queries, while the performance of the two policies is comparable for small result queries. It also showed that result distribution buffering offers performance improvements for large result queries while not providing any benefits for small result queries. Based on these experiments, we are going to use distributed result distribution with opportunistic result-reading for unordered queries as the default policies in AsterixDB. We will turn-off result distribution buffering by default, but will expose a configuration parameter to users so that they can enable it depending on the system's deployment and expected workloads.

While the result distribution framework in AsterixDB already caters to the result management needs of a Big Data system and does reasonably well at it, it still leaves room for improvements. Let us look at some of those possible improvements. The current implementation of result life-cycle management uses fixed size datastructures to recycle the old query results. This can be replaced by a better life-cycle management mechanism. Since the results for synchronous queries cannot be read by the client more than once, they can be cleaned up as soon as the clients read the results first time. One the other hand, the results for asyn-

chronous queries can be kept around for a fixed amount of time before cleaning them up. This duration can be exposed to the users as a configuration parameter. Another area that leaves room for improvement or experimentation is the page replacement policy for deciding which result frames to store in memory and/or on disk. The current implementation uses a least recently active jobs based mechanism to find the victim job whose memory frame is flushed to disk. We can experiment with alternative page replacement policies such as keeping the result frames in memory for the jobs whose results were most recently produced since they have higher chance to be read sooner or a policy which gives every job a fixed number of memory frames to store their results so that smaller result queries have higher chance of keeping their whole result set in memory, thereby offering memory speed performance for small result queries.

Bibliography

[1] V. Ayyalasomayajula. HERDER: A Heterogeneous Engine for Running Data-Intensive Experiments & Reports. *M.S. Thesis, Computer Science, University of California, Irvine*, 2011.

[2] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras. ASTERIX: towards a scalable, semistructured data platform for evolving-world models. *Distrib. Parallel Databases 29, 3*, June 2011.

[3] V. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing. *International Conference on Data Engineering (ICDE)*, 2011.

[4] V. R. Borkar, M. J. Carey, and C. Li. Big Data platforms: what's next? *ACM Crossroads 19(1)*, 2012.

[5] V. R. Borkar, M. J. Carey, and C. Li. Inside "Big Data Management": Ogres, Onions, or Parfaits? *Extending Database Technology (EDBT)*, 2012.

[6] Hadoop. `http://hadoop.apache.org/`.

[7] JDBC. `http://en.wikipedia.org/wiki/Java_Database_Connectivity`.

[8] TPC-H website. `http://www.tpc.org/tpch/`.