# Laasie: Towards One-Size-Fits-All Database (OSFA) Architecture

by

Ankur Upadhyay

The thesis submitted to the faculty of the
University at Buffalo, The State University of New York
In partial fulfillment of the requirements for the degree of

MASTER IN SCIENCE
In
Computer Science and Engineering

Dr. Lukaz Ziarek
Dr. Oliver Kennedy
Dr. Bina Ramamurthy

June, 2014

**ABSTRACT**

Variation in application workloads, requirements, and hardware has often leaded to claim that One-Size-Fits-All databases are dead, as they require precisely adjusting a wide range of parameters to be competitive. Consequently, several specialized data management systems have emerged in past several years. Applications with continuously emerging requirements and features require maintaining different data management systems, which often prove costly to companies and enterprises in terms of development, maintenance and DBA costs.

The benefits of having a single interface, to work with, are quire compelling: Client applications can be prototyped quickly, and infrastructure tuning and development can be deferred until the applications runtime characteristics are better understood. Consequently, a slew of “database building blocks” have arisen, allowing users to custom-tailor a common core engine to their application’s requirements. The thesis presents “Abstract Syntax Tree” (AST) as a database building block. Our building block framework, called Abstract Syntax Tree database, represents data state with the sequence of updates, stored in the form Abstract Syntax Trees. It provides a simple rewrite-rule-based interface, which allows extremely fine grained control over a wide variety of database management design aspects, from layout and indexing, through client side caching behavior. The thesis also presents Laasie, a document store Abstract Syntax Tree database. Using Laasie, we demonstrate the feasibility of AST-Databases in general, and then show how Laasie effectively supports a complex use case with highly variable workloads: collaborative web applications.

**ACKNOWLEDGEMENTS**

This project would not have been possible without the continuous support of many people. I would like to express my special appreciation and thanks to my advisors, Dr. Lukaz Ziarek and Dr. Oliver Kennedy. I like to thank you for continuously supporting me during my research and taking some time off your busy schedule to go over several revisions of my thesis and make some sense out of the confusion. I would also like to thank my committee member, Dr. Bina Ramamurthy, who offered her guidance and support. I would also like to thank you for your brilliant comments and suggestions and letting my defense be an enjoyable moment. I would also like to thanks my project mates, Sumit, Daniel, Palanippan, Nikhil, Subrahnil, Kaushal and Shail, for making this journey beautiful and enjoyable.

I would also like to express my gratitude to my parents, Dr. S. K. Upadhyay and Mrs. Manju Upadhyay, for always mentally supporting me and motivating me to achieve my aims and goals. My final acknowledgement is to my sister, Neha Upadhyay, and my brother-in-law, Mr. Peeyush Agrawal, for always being with me and making me smile during the hard times.

# Contents

# Chapter 1

# Introduction

In the past several years we have witnessed a split in data management systems creating several specialized solutions [1, 25, 38, 13, 21, 9, 36]. Relational databases were prominent over decades, when the query flexibility and the data relationships are of utmost importance. With an increase in the amount of unstructured data, where data relationships have been pushed to the back seat, the database community introduced several "NoSql" [43] databases, that have flexible schemas and support unstructured data. Each data management system in this class is specialized to handle a specific category of workloads.

Today an application's features changes quickly and requirements evolve with time. With such evolving application and workload characteristics, the maintenance of several specialized database systems can prove costly in terms of development and maintenance. Moreover, it is very difficult to predict application runtime characteristics at an early stage making it harder for developers to choose the appropriate database management system for their application. An alternative solution is to provide a tunable One-Size-Fits-All (OSFA) database that can be tuned according to application requirements and characteristics. The tunable infrastructure allows developers to quickly prototype client applications, and tune and develop the infrastructure only when the application's runtime characteristics are better understood. However, an infrastructure which exposes an increasing amount of tuning options is not an answer since it is neither scalable nor practical for developers [42].

This observation has lead to another strategy to provide "database building blocks" to application developers, with which they can build application-specific database engines. The developers can customize the database engines over time, which is best suited for varying application and workload characteristics. This

allows developers to focus on the client interface of the application, only requiring them to specialize the infrastructure once the workload characteristics becomes more clear and better understood.

For building database engines from building blocks, application developers require very fine grained control over the infrastructure components. The thesis present the new class of database building blocks called "Abstract Syntax Trees" (ASTs) that provides fine grained control over infrastructure components to application developers. An AST is a tree-like structure which represents the insert/update/delete operations performed over application state variables. The infrastructure, instead of storing the computed (or reified) value of data components, stores a sequence of uncomputed updates, represented as ASTs. The reified value of the state variables can be obtained by evaluating the sequence of updates. We call this infrastructure an "AST databases" (ADBs).

ASTs, much like the data manipulation language from which they are generated, are amenable to program analysis techniques [46, 24]. This allows developers to fetch application and workload characteristics at runtime by analysing ASTs. With application runtime characteristics in hand, developers will be in better position to make important tradeoffs during application runtime and frame application specific optimizations over the stored ASTs. As an example, an AST database gives full control to user over if, when, and where the evaluation of ASTs occurs. For update heavy workloads, updates can be made faster by delaying the ASTs evaluation at the time of the read operations. Similarly, for read heavy workloads, the ASTs can be evaluated at the time of update operations, similar to traditional databases. Moreover the ASTs evaluation tasks can be computed at the server side or can be offloaded to the client side. This decision can be based on several factors like network bandwidth, server load, client computation capacity, etc.

One motivating scenario for AST databases is collaborative web applications like Google Docs [22], Office 365 [15], or Dropbox [18], where an AST database's dynamic specialization capability can be fully illustrated. Collaborative applications allow multiple users to share and edit a single application state. These applications requires extensive infrastructure support, first to relay/updates between users, and second to persist application state. The single application may require different infrastructure behavior for different components of its state. For example, a chat system in application may be best served by the log based approach, where the text written by users get appended to the log, whereas the images may better be stored as raw entity. Application developers often deploy new features and functionality, resulting in frequent, live changes to the application's work-

load, schema, and data management needs. This forces application developers to use hybrid infrastructures cobbled together from multiple distinct data management systems. This variability in a collaborative application's requirements and characteristics makes AST databases an ideal choice. AST databases provides application developers fine grained control over the exact way the application state is stored and processed. Simultaneously, a single, consistent interface allows the front end to be developed independently of any infrastructure optimizations. This allows for rapid prototyping and early experimentation to obtain detailed workload characteristics before the infrastructure is specialized.

# Chapter 2

# State of the Art

Variation in application workloads, requirements, and hardware has lead to claims that One-Size-Fits-All (OSFA) databases are dead [42], as they require precisely adjusting a wide range of parameters to be competitive. In past several years, many specialized solutions [1, 25, 38, 13, 21, 9, 36] have been emerged. By exposing a limited or specialized API, the system can make strong assumptions about the application workload. A tighter coupling with the applications workload allows the data management system to provide improved performance, scalability, and ease-of-management.

The chapter starts with describing the various categories of database systems in today's database industry. The following sections discusses the problems arising with several database systems management, and the approach taken by industry and academia to deal with those problems.

## 2.1 Database systems

Several categories of database systems have emerged to resolve some of the key problems confronted by the traditional relational databases. Storing huge amounts of data (often termed "Big Data") for faster access is a problem in the forefront. With the evolution of large sets of unstructured data, data relationships have been pushed to the back seat, prioritizing faster data access. Several classes of database systems have been developed which have flexible schemas and give less importance to data relationships. Such database systems can be classified into four main categories: key-value databases, wide-column databases, document databases and graph databases. Furthermore, several hybrid implementations that combine two

of these models are also available.

Each solution is different and each performing better under certain workload conditions. Although there are other important factors which influence the choice of database solution, performance is a metric of paramount importance because one of the key promises of these solutions is a fast data access.

### 2.1.1 Document Databases

**Examples: MongoDB, ArangoDB, BaseX, Informix, MakeLogic, etc.** Document databases are developed to query data without having precise knowledge of its structure. They are designed for storing, retrieving, and managing document-oriented information, also known as semi-structured data. Fundamental to document databases is the concept of data entities named 'documents'. They are analogous to table rows in RDBMS, but are less rigid. Each document is associated with a unique key, which is a simple string, uniform resource locator (URL), or a path. A document holds its data in the form of an attribute-value set. Attributes can differ across documents, enabling flexible schema. Documents are independent and hence improves performance and decreases concurrency side effects. Document databases are described using JSON, XML or derivatives.

### 2.1.2 Wide Column Databases

**Examples: Cassandara, Hadoop, HBase, etc.** Column oriented databases support high performance applications in the business intelligence domain such as data warehouses, customer relationship management (CRM) systems, and other ad-hoc inquiry systems where aggregates are computed over large number of similar data items. Column oriented data management systems store data tables as sections of columns of data rather than as rows of data (as stored in traditional relational databases). Column oriented databases are more efficient than traditional row-oriented databases on workloads consisting of column oriented queries (i.e. querying/modifying/inserting a notably smaller subset of all columns of data). However, column oriented databases would not be as efficient as traditional row oriented databases on the workloads which involves operations like insertion and extraction of entire rows of the data, like inserting a new row or extracting all columns of single row. In a nutshell, column oriented databases are well-suited for OLAP-like workloads (e.g. data warehouses) which typically involve a smaller number of highly complex queries over all data (possibly terabytes). However,

they would not be well suited for OLTP-like workloads which are more heavily loaded with interactive transactions.

### 2.1.3 Key-Value Databases

**Examples: Membase, Riak, Redis, etc.** This class of database is very popular for web applications. Key-Value Databases are schema-less databases which store data as values associated with their respective keys. This class of databases is designed for heavy read workloads and for handling fluid data types that cannot be easily tabularized. Being a schema-less database, key value databases allow storage of arbitrary data that are indexed using a single key, and allow efficient retrieval. However, because they store unstructured data, key-value databases are not as efficient when searching for a particular subset of data. One needs to either read every record or build secondary application specific indexes.

### 2.1.4 Graph Databases

**Examples: Neo4j, etc.** Graph Databases are very popular for social networks where complex, many-to-many relationships need to be set up. This type of data relationship is hard to be implemented in most of the other data management systems. A graph database can be defined as the structure which consists of nodes, edges, and properties to represent and store data. Nodes are analogous to records in RDBMS. Edges define the relationship between different graph nodes or data. Graph Databases are most efficient when dealing with associative and related data. Graph databases are highly scalable and can easily manage ad-hoc data. In general, graph databases allow developers to work with an object-oriented, scalable network instead of fixed schema.

The table below summarizes the key classes of specialized database solutions.

| Type of Database | Examples | Specialized for | Disadvantages |
|---|---|---|---|
| Document Databases | MongoDB, ArangoDB, BaseX, etc. | Querying semi-structured data. Heavy read workloads | Not suitable for Update/Insert heavy workloads since it requires updating/inserting in every document maintained |
| Column Databases | HBase, Hadoop, etc. | Querying/Modifying a notably smaller subset of all columns of data | Not suitable for queries involving all columns of a row. |
| Key-Value Databases | Riak, Redis, Membase, etc. | Heavy lookup workloads, continuous data reads and writes | Searching for particular record is ineffecient |
| Graph Databases | Neo4j, etc. | Workloads involving many-to-many relationships | Difficult to scale horizontally. |

## 2.2 OSFA (one-size-fits-all) Architecture

Each database design is motivated to handle a category of workloads arising from specific use cases. This design consists of several trade-offs which make them highly efficient for some categories of workloads but inefficient for others. For instance, column store databases, are especialy effecient for OLAP queries, but inefficient for OLTP queries. As a consequence, today's companies and enterprises have to manage and integrate several types of data management systems. Sometimes the presence of different types of data management systems, rather than making the data management easier, proves to be harder and costlier to companies. Many application developers have to spend engineering effort on database systems management tasks like copying data from one database to another. This leads to extra costs in terms of development, maintenance and DBA cost. A single data management system which covers multiple database use cases would be able to overcome these issues.

Specialized database engines are expected to outperform the traditional RDBMS (attempting to be one size fits all solution) in several application areas, including Data warehousing, Stream processing and scientific and intelligence databases. Variation in application workloads, requirements, and hardware often makes OSFA architectures unsuitable and inefficient. However, there is something very ap-

pealing about tunable OSFA architectures: client applications can be prototyped quickly, and infrastructure tuning and specialization can be deferred until the application's runtime characteristics are better known and understood.

A single tunable infrastructure is not an answer, since an infrastructure that simply exposes an ever increasing amount of tuning options, is neither scalable nor practical for developers [42]. An alternative approach of providing "database building blocks" to developers, has often been employed [12, 17]. From these core components, developers can build customized database engines that are specialized to their application's requirements. Such systems allow developers to focus their efforts on client applications first, only specializing the infrastructure post-hoc, as the client application matures and its workload characteristics become better understood. Often, this specialization can be done without substantive changes to the application's existing client/server interface-an especially crucial feature for domains of web applications, where workload characteristics change frequently.

### 2.2.1 Related Work

The "building block databases'' can be categorized into two categories: (1) Language based building block databases, and (2) Log based building block databases.

#### Language Based Building Blocks

Packaging all database technology into a single unit of development, maintenance, deployment and operation is not an answer since such an infrastructure will not be appropriate for variation in application workloads experieced by today's applications [42]. Instead, an alternative approach of providing RISC-style, specialized data managers have often been employed [12]. Such data management systems are often have limited functionality. Low-level RISC style database architectures provide target applications and end-users with a sufficient, but minimal feature-set, thus avoiding the overheads of supporting unneeded functionality. More recent efforts have begun to realize this vision, ranging from general approaches that exploit programmable and/or algebraic commutators to minimize locking overheads [40, 19, 14, 16] to full data management systems. The MonetDB [9] and LogicBlox [23] systems both express database accesses and manipulations through low-level, fine-grained, algebraic instruction sets (MIL/X100 and Datalog/LogiQL, respectively), allowing their respective engines to optimize execution around a precisely characterized set of required capabilities. For fixed workloads,

DBToaster [29] and HyPer-LLVM [34] instead perform aggressive workload compilation, producing minimal, highly-efficient data processing engines.

Recent efforts on compilers for Domain Specific Languages can also be thought of as belonging to this class of building-block databases. Efforts to optimize DSLs range from early work on collection programming and object stores [7, 47], to more recent efforts directed at the compiler pipeline. Compiler toolchains such as Delite [11] or K3 [41] allow end-users to easily write compilers for domain-specific data processing languages. By providing developers with a set of core primitives and a set of universal equivalences and rewrite rules for these primitives, the toolchain makes it easy to bootstrap a compiler that has reasonable performance out of the box. Developers can then provide additional equivalences and rewrite rules based on domain knowledge to further improve the performance of compiled code.

### Log Based Building Blocks

The second class of building block databases uses logs as building block. This strategy allows for an extremely low-cost write primitive that still ensures ordering. It then falls to the infrastructure developer to reconstructure, organize, or persist the log to ensure the appropriate tradeoffs for performance [17], data-avaliability [3], consistency [28], or replication [8, 6, 48, 44].

**OctopusDB [17]:** In OctopusDB all data is collected in a central log, i.e. all insert and update operations create logical log-entries in that log. Based on that log, it may then define several types of optional Storage Views (or materialized views). A Storage View (SV) represents all or part of the log in a different physical layout. The Storage view selection is solely based on the workload in hand. This single abstraction converges the problems of query optimization, view maintenance, index selection, as well as storage selection into a single problem of storage view selection.

Though coarse-grained, views give end-users a remarkable level of control over how and when computation occurs. The cost of appending to the main log is minimal, placing the full burden of computation on subsequent read operations. By instantiating views, users increase the computational burden on write operations (as the views must be maintained), but simultaneously reduce the cost of some reads. The mechanic of views gives end-users an intuitive way to control how and when the engine performs computation on the data.

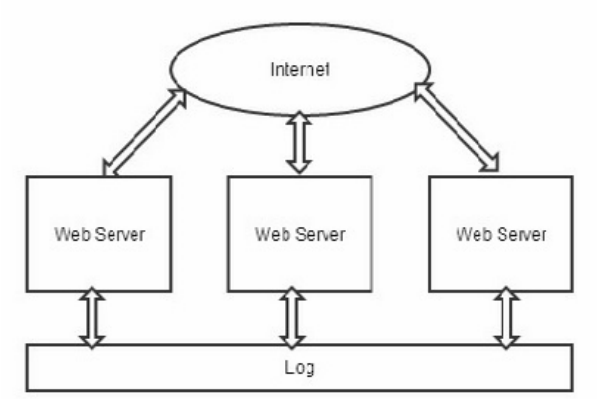



Figure 2.1: The Hyder Architecture.

**Hyder [8]:** Hyder is an another example which uses the log based abstraction to allow scaling out without partitioning the database or application, making it well suited for data center environment. Hyder consists of a log-structured multi version database, stored in a flash memory and shared by many multi-core servers over a data center network as shown in Figure 2.1. Since it uses a data sharing architecture (i.e. the main log), all servers can read from and write to the entire database, which in turn simplifies the application design by avoiding partitioning, distributed programming, layers of caching and load balancing. Also, in Hyder, update transactions execute on one machine and write to one shared log. Hence, it does not require two-phase commit which saves message delays.

# Chapter 3

# AST Databases

The chapter introduces Abstract Syntax Trees (AST) as a "database building block". AST databases (ADBs) do not store the computed value (or reified value) of the application state variables. Instead it stores a sequence of un-evaluated updates, applied over application state variables. The reified value of the state variables can be obtained by collecting all the updates and evaluating them. The un-evaluated updates are expressed in a data manipulation language built over AST database and are stored within infrastructure in the form of Abstract Syntax Trees (ASTs). An abstract syntax tree is a tree-like structure which represents the update performed over a state variable. The abstract syntax trees, much like the data manipulation language, from which they are generated, are amenable to the program analysis which allows developers to fetch workload or application runtime characteristics from it. AST databases provide developers fine grained control over infrastructure components, which helps them to make important decisions and application specific tradeoffs. As an example, developers have the full control over if, when, and where the evaluation of AST should occur.

This chapter starts with highlighting the motivation behind using the AST as a "database building block" and how their amenability to program analysis helps to fetch application-specific characteristics, which can later be used to build application-specific optimizations over an ADB infrastructure. The chapter mainly focuses on the structure of AST databases and how ASTs are stored within its infrastructure, and how the AST database's primary operations (i.e. update, read, and evaluation) are performed over its structure.

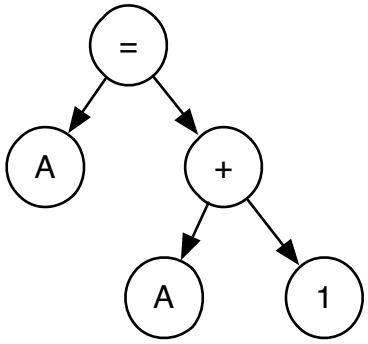



Figure 3.1: The Abstract Syntax Tree (AST) of update ‘A=A+1’.

## 3.1 AST: Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a tree-like structure which is generated by analysing the syntax of the update represented in the data manipulation language built over it. Consider the update ‘A=A+1’ which increments the value of state variable ‘A’ by 1. The update can be represented as abstract syntax tree as shown in Figure 3.1.

### 3.1.1 AST: Database Building Block

Chapter 2 discussed several building block databases. These databases are categorized into two main categories (1) Log-Based Building block databases [17, 8, 6], and (2) Language based building block databases [9, 23, 29, 11].

OctopusDB [17] (a log based building block database) stores a history of updates in the central log and allow developers to initalize different materialized views, each supporting the specific workload’s requirements. Although this approach reduces the complexity of creating custom databases, it provides coarse-grained control over the infrastructure components. As an example, OctopusDB provides user control of how and when computation of updates should happen. The computation can be deferred until reads and makes writes extremenly effecient, since it only requires the minimal cost of appending the update to the central log. On the other hand, if the user defines materialized views, it reduces the cost of read operations making writes computational heavy. Developers have to make an important trade-off according to the application workload. Also, it may cause the unneccessary duplication of data by keeping a copy of it in central log and the

materialized view created over it.

On the other hand language based building block databases [9, 23, 34, 29, 11], focuses on providing domain specific data processing languages which allows users to easily write workload specific compilers powered with rewrite rules, which offers the performance comparable to the specialized solutions. The major advantage of the language based building blocks is that they are amenable to the post-hoc optimizations. With the evolution of application requirements, the users can provide additonal equivalences and rewrite rules based on domain knoweldge to further improve the performance. Although, these databases provides enough fine grained control over the data storage layer, they focuses on providing application developers with a sufficient, but minimal subset of functionalities. Also the focus of these systems is on data-processing and not persistence.

We propose Abstract Syntax Trees(ASTs) as a "database building block". An AST database stores a sequence of un-evaluated updates, represented as ASTs. Encoding together the ASTs combines the best festures of language and log based building block databases. Like language based building blocks, ASTs provide users fine grained control over how the data is stored and accessed by allowing users to develop the equivalences and rewrite rules. Instead of specifying the rewrite rules in terms of domain specific data processing language, the user specify them in terms of ASTs which can be coupled with any language and thus does not limits the infrastructure to specific category of workloads. Moreover, representing update history as a sequence of ASTs, provide users much finer grained control over stored data, unlike log based building block databases. As an example, rewrite rules on ASTs provide users much fine grained control over how, when and where evaluation should happen. An ADB can evaluate ASTs at the time of update, or can delay it till the time of read operation, or when the infrastructure is idle. It provide developers the full control over when to trigger the evaluation, thus helping them to make important decisions and tradeoffs during application runtime.

### 3.1.2 Analysing ASTs

ASTs, much like the data manipulation language,from which they are generated, are amenable to program analysis techniques [46, 24]. By analysing ASTs, important information about an application update can be extracted, including:

1. AST write set: The write set of an AST include state variables which are updated/modified (or initialized) by the operation performed. Given an AST

($\hat{A}$), the function $AST_W$ extracts the write set.

2. AST read set: The read set of an AST include state variables on which AST depends i.e. the state variables whose reified value is required to evaluate the AST. Given an AST ($\hat{A}$), the function $AST_R$ returns the read set.

The functions $AST_W$ and $AST_R$ are called metadata extraction functions. The implementation of metadata extraction functions is connected to the semantics of the data manipulation language built over an AST database's infrastructure.

**Example 1** *Consider the AST shown in the Figure 3.1. The write set of the AST includes state variable 'A' since it is been updated by the AST. The read set of AST also includes the state variable 'A' since its previous value is required to evaluate the AST.*

## 3.2 ADB Structure

In its most general form, an AST database, instead of storing the reified value of state variables, stores an ordered sequence of un-evaluated updates made over application state variables. The individual update is represented in the form of an AST. Thus the reified value is never stored in an ADB infrastructure, but the same can be obtained by evaluating the sequence of ASTs. The AST may depend upon other ASTs. To evaluate an AST, the infrastructure should be able of fetch its dependent ASTs from the stored sequence. Thus an AST stored within the database, should have the sufficient knowledge about the position (in the ASTs sequence) of ASTs on which it depends. This observation leads us to store ASTs and their dependencies in a data structure, similar to graph, with each AST stored in the graph node pointing to its dependent ASTs by outgoing edges. In an ADB-ecosystem, such a graph structure is called "Data Flow Graph (DFG)".

A node of the data flow graph stores the AST of an update performed over a state variable. An ADB maintains the ordered sequence of the nodes in the order of their creation. Each node in the DFG is labelled with the positive integer, which specifies their specific position in the ordered sequence.

An ADB's DFG is denoted as DFG = (V, E) with

- V is ordered sequence of nodes in a DFG. A vertex v = ($\hat{A}$, n, Adj(v)), in the sequence 'V', is considered to be a DFG node storing the AST $\hat{A}$ and having sequence number 'n'. 'Adj(v)' is the associated adjacency list which is an unordered list containing node's direct neighbors.

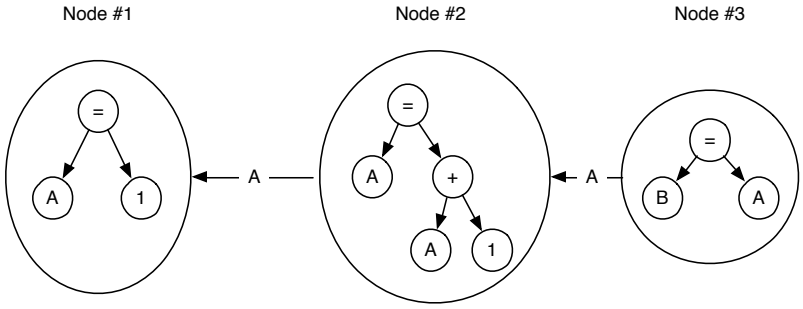



Figure 3.2: An ADB's Data Flow Graph.

- E is the set of ordered pair of DFG nodes called edges. An edge e = $(x, y)^z$ is considered to be a directed edge from vertex x to vertex y. The AST stored in node x is said to be dependent upon the AST stored in node y. The dependency between the two ASTs is denoted by z.

**Example 2** *An example of an ADB's DFG is shown in the Figure 3.2. The DFG consists of three nodes, each containing the AST of an update performed on database state. The directed edges in a DFG denote the dependencies between connected ASTs. The reified value of state variables can be obtained by evaluating the ASTs of updates performed over them. For example, the value of the state variable 'A' can be obtained by evaluating the ASTs stored in node #1 and node #2 (of Figure 3.2). Similarly, the value of the state variable 'B' can be obtained by evaluating the AST stored in node #3. Since the AST depends upon 'A', evaluating value of 'B' requires first evaluating the reified value of 'A'.*

An ADB exposes two primitive operations:

1. The Data Flow Graph (DFG) may be queried by program analysis such as dependency analysis, slicing [46], and similar techniques [24].

2. The sequence of ASTs, or Data Flow Graph (DFG), may be modified (a new AST can be added or the several ASTs can be combined together) through deterministic, piecewise graph transformations [27, 32, 2].

### 3.2.1 ADB Reads

To obtain the reified value of state variables, the ASTs stored in an ADB's DFG needs to be computed. Collecting and evaluating all ASTs, when only a particular

state variable (or subset of state variables) is required, is unnecessary and inefficient. However, in order to obtain the full reified database state requires evaluation of all stored ASTs. In order to make the read operations efficient, ADB relies on the program analysis technique called program slicing. Program Slicing reduces the amount of computation needed to obtain the reified value of state variables.

### Program Slicing

Program slicing is a common and powerful compiler technique which can be used for program analysis and optimizations. Program slicing is the computation of the set of program statements, the program slice, that may affect the values at some point of interest, referred to as a slicing criteria. Intuitively, given a sequence of instructions and a slicing criteria, a slice is is the minimal subset of instructions that is still guaranteed to produce equivalent values for the specified outputs.

**Example 3** *For example, consider the simple sequence of updates:*

1. *A = 1*
2. *B = A + 2*
3. *C = A + 3*
4. *D = B + C*

*If the user intends to read the state variable 'C', the slice with respect to 'C' can be computed to return computations 1 and 3.*

### Computing Program Slice on ADB's DFG

Computing a slice, with respect to a particular state variable, on an ADB's DFG requires two steps:

1. Finding the DFG node storing the AST of the latest update performed over the state variable.
2. Finding the set of descendant nodes of the node fetched in the previous step. The descendant nodes contains all the ASTs which are required to obtain the reified value of the state variable.

Given an ADB's DFG and slicing criteria variable (A), function $READ_V$ iterates over the ordered sequence of nodes in reverse order and finds the first node containing the update's AST performed over the state variable, desired to be read. Once the appropriate node is fetched, function $READ_D$ applies Breadth First Search [20] on it, to collect all the descendent nodes. The function $READ_D$ eliminate the duplicate ASTs (from the set of descendent nodes) encountered during the traversal. Also, the function $READ_D$ uses a priority queue to temporarily store the DFG nodes during traversal. Since the DFG nodes are ordered in the order of their creation, the use of a priority queue [33] ensures that the descendant nodes are collected in the same order while traversal.

**Example 4** *Consider the following sequence of the updates performed over the application state variables:*

1. *A=1*
2. *B=A+1*
3. *A=A+1*
4. *B=B+1*
5. *C=A*
6. *D=B+C*

*The sequence of updates are stored in an ADB's DFG, as shown in Figure 3.3. The DFG contains six nodes, each containing the AST of the computation performed. Suppose if the user intents to collect all the updates performed over state variable 'B', the function $READ_V$ will iterate over the ordered sequence of DFG nodes in reverse order and select the node containing the computation 'B=B+1', since it is the latest operation performed on 'B'. Function $READ_D$ will then follow the selected node's outgoing edges to collect the descendant nodes. The two functions together will select the nodes containing the computation 'B=B+1', 'B=A+1', and 'A=1'. The three ASTs collected can then be evaluated to obtain the reified value of state variable 'B'.*

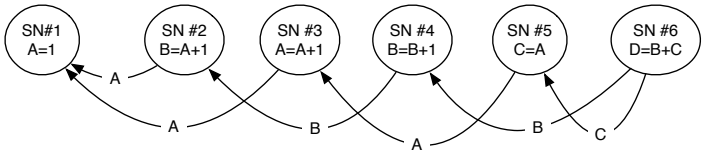



Figure 3.3: An ADB's DFG

**Algorithm 1** $READ_V$(DFG(V, E), A)

**Require:** `DFG(V, E)`, An ADB's DFG.
**Require:** `A`:$A$, The state variable desired to be read.
**Ensure:** `rootNode`:$rootNode$, The DFG's node containing the latest update's AST over application state variable, A.

```
for node ∈ Reverse(V) do
    component ← AST_W(node.Â)
    if component == A then
        rootNode = node
        Break
    end if
end for
```

**Algorithm 2** $READ_D$(DFG(V, E), sourceNode)

**Require:** `DFG(V, E)`, An ADB's DFG.
**Require:** `sourceNode`, The source node from which BFS is to be applied.
**Ensure:** `descendants`, The set of descendants of the selected root nodes.

```
Create Priority Queue Q.
Enqueue sourceNode onto Q
while Q ≠ Empty do
    node ← Dequeue(Q)
    for adjNode ∈ Adj(node) do
        if adjNode not in Q then
            descendants = descendants ∪ adjNode
            Enqueue adjNode onto Q
        end if
    end for
end while
```

**Evaluation** The ASTs collected from functions $READ_V$ and $READ_D$ are evaluated to obtain the reified value. Because an ADB stores the ordered sequence of update ASTs, user has the full control to decide where, when and how to evaluate the collected ASTs. The evaluation can happen at the server side, or can be offloaded to the client side. The user can also decide when to perform at the evaluation. It can be performed at the time when update has been applied to database, or when the user reads from the database, or when the infrastructure is idle (i.e. not processing any update or read requests). Moreover, the evaluation can be partitioned such that part of the evaluation happens at the time of update and part of it happens at the time of read. This decision can be based on several factors such as server load, network bandwidth, client computation capacity, workload specifications, etc.

### 3.2.2 ADB Updates

In the ADB ecosystem, an update to an application state variable represented as an Abstract Syntax Tree, is appended to the stored sequence of ASTs. This changes the existing DFG's state. Since a sequence of ASTs is stored in the DFG, an update to an ADB requires two steps:

1. Create a new DFG node to store the update's AST.
2. Connect the new node with existing DFG nodes to track any AST read dependencies. Recall that the AST read set can be obtained by metadata extraction function $AST_R$. The new node created in the previous step is connected to each node containing the latest update to a read dependency (of the new AST) by an outgoing edge.

When an update represented by Abstract Syntax Tree $\hat{A}$ is applied to ADB infrastructure, it transforms its DFG(V, E), to the state represented by $\hat{D}FG(\hat{V}, \hat{E})$ with

- $\hat{V} = V \cup UPDATE_V(\hat{A})$
- $\hat{E} = UPDATE_E(DFG(V, E), \hat{A})$

The function $UPDATE_V$ creates a new DFG node that stores the new update's AST ($\hat{A}$). Once the new DFG node is created, the function $UPDATE_E$ creates an edge between the new node and the existing nodes which contain the latest writes for the new AST's read dependencies. The nodes of an ADB's DFG are labeled

with the sequence number, which specifies the order in which the AST is appended to the database state. The function $UPDATE_E$ iterates over the sequence of the nodes in reverse order til it finds the first node containing an update on the new AST's read dependency. The node found contains the latest update over the AST dependency. The directed edge is created from new AST node created by function $UPDATE_V$, to the node found by function $UPDATE_E$. The process is repeated for every AST dependency. The function $UPDATE_E$ requires $O(m*n)$ time, where m is the number of AST read dependencies and n is the number of DFG nodes.

**Algorithm 3** $UPDATE_E$(DFG(V, E) $\hat{A}$)

**Require:** `DFG(V, E)`, The current ADB's DFG.
**Require:** $\hat{A}$, The AST of the new update.
**Ensure:** $\hat{DFG}(\hat{V}, \hat{E})$, The modified DFG

```
newNode ← UPDATE_V(Â)
V̂ = V ∪ newNode
dependencies ← AST_R(Â)
for dep ∈ dependencies do
  for node ∈ Reverse(V) do
    component ← AST_W(node.Â)
    if dep == component then
      Break
    end if
    Ê = E ∪ E(newNode, node)^dep
  end for
end for
```

**Example 5** *An example of an ADB update is shown in Figure 3.4. The initial ADB stage consists of three nodes each containing the ASTs of the computations performed on the database. Consider an update 'A=A+2' performed on ADB infrastructure. The metadata extraction functions, $AST_R$ and $AST_W$ will be applied on update's ADB to extract the read (i.e. A) and write set (i.e. A) of the AST respectively. Function $UPDATE_V$ will create the new graph node to store the update AST. Function $UPDATE_E$ will connect the newly created node to the DFG node containing operation 'A=A+1', since this is the latest computation performed on the read set of the AST (i.e. A).*

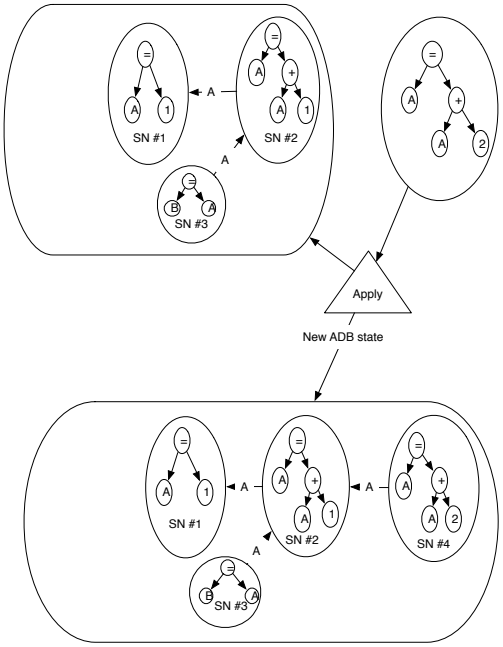



Figure 3.4: An ADB's DFG transformation after an update operation.

## 3.3 Optimizations over ADB's DFG

This section describes some of the optimizations which can be built over an ADB's DFG to make its primary operations (read, write, and evaluation) efficient.

### 3.3.1 Indexing: Faster reads and writes

Recall that an ADB maintains its sequence of DFG nodes in the order of their creation. In order to process update and read requests, the infrastructure iterates over the ordered sequence of nodes in reverse order. To append a new update, the function $UPDATE_E$ iterates over the DFG node sequence in reverse order to find the nodes containing the latest update performed over the new update's dependencies, if any. Similarly, to process the read requests, the function $READ_V$ iterates over the DFG node sequence to find the node containing the latest update performed over the application component, desired to be read.

The update and read functions can avoid iterating over the sequence of DFG nodes with the help of a special DFG node called an Index Node. An Index node is a Hash Map (a structure that maps keys to values) built over an ADB's DFG. The keys are application state variables. The values are the DFG nodes that contains the latest update performed respective keys.

#### Faster Reads

The read function $READ_V$ can avoid unnecessary iteration over the DFG nodes by using index nodes. The improved process is illustrated in the function $READ-INDEX_V$. Function $READ-INDEX_V$ requires $O(1)$ time to fetch a node through an Index node, as opposed to the function $READ_V$ which requires $O(n)$ time for iterating over all dependencies (where n is the number of dependencies). Once the DFG node containing the latest update is fetched, the function $READ_D$ can be used to collect the descendant nodes in DFG.

**Example 6** *Consider the ADB's DFG of the ADB-Read example in the Section 3.2.1 with the presence of an index node as shown in Figure 3.5. If the user intents to read state variable 'B', the function $READ-INDEX_V$ uses the index mappings to select the node containing the AST of update 'B=B+1'. Once a node is selected, function $READ_D$ collect all the descendant nodes. Functions $READ-INDEX_V$ and $READ_D$, collect the ASTs of computations 'B=B+1', 'B=A+1', and 'A=1'.*

**Algorithm 4** $READ-INDEX_V$(DFG(V, E), A, $\hat{I}$)

**Require:** `DFG(V, E)`, An ADB's DFG.
**Require:** `A`:$A$, The state variable desired to be read.
**Require:** $\hat{I}$, DFG's Index node.
**Ensure:** `rootNode`:*rootNode*, The DFG node containing the latest update over the state variable desired to be read.
`rootNode` = Index-Map($\hat{I}$).get(`A`) #
fetch the DFG node from the index mappings (Index-Map)

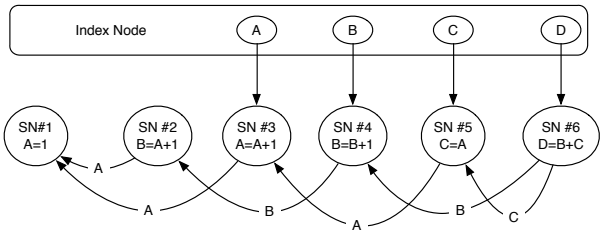



Figure 3.5: An ADB's DFG with Index Node.

### Faster Updates

The update function $UPDATE_E$ avoids unnecessary iteration over ordered sequence of DFG nodes, with the help of index node which maps the application component to the DFG node containing the latest update performed over the respective application component. The new function $UPDATE-INDEX_E$:

1. Uses metadata extraction function $AST_R$ to fetch the read dependencies, if any.

2. Uses mapping maintained by Index node to fetch the nodes containing the latest updates performed over the read dependencies. The Index node fetches the required DFG nodes in O(1) time as opposed to iterating over the sequence of nodes which requires O(n) time, where n is number of nodes in DFG. The overall time complexity of $UPDATE-INDEX_E$ is $O(m)$, as opposed to $UPDATE_E$, whose time complexity is $O(m*n)$ (m is the number of read dependencies of new update and n is the number of DFG nodes).

The state variable used as an index key can be fetched from the metadata extraction function $AST_W$. If the index has been initialized for this key, the new entry has to be inserted in the Index node mappings. If the existing component has

**Algorithm 5** $UPDATE-INDEX_E$(DFG(V, E), $\hat{A}$, $\hat{I}$)

**Require:** `DFG(V, E)`, The current ADB's DFG.
**Require:** $\hat{A}$, The new update's AST
**Require:** $\hat{I}$:*indexNode*, DFG's Index node.
**Ensure:** $DFG(\hat{V}, \hat{E})$, The modified DFG
  `newNode` $\leftarrow UPDATE_V(\hat{A})$
  $\hat{V} = V \cup$ `newNode`
  `deps` $\leftarrow AST_R(\hat{A})$
  `indexMap` $\leftarrow$ Index-Map($\hat{I}$)
  **for** `dep` $\in$ `deps` **do**
    `node` $\leftarrow$ `indexMap(dep)`
    $\hat{E} = E \cup E(\texttt{newNode}, \texttt{node})^{\texttt{dep}}$
  **end for**

been updated, the respective entry in Index node mapping has to be modified to point to the new update since it is now the latest update over respective component. The function $UPDATE-INDEX_I$ makes the appropriate changes to the Index node.

**Algorithm 6** $UPDATE-INDEX_I$($\hat{A}$, $\hat{I}$)

**Require:** $\hat{A}$, The AST of the new update.
**Require:** $\hat{I}$, DFG's Index node.
  `newNode` $\leftarrow U_V(\hat{A})$
  Index-Map($\hat{I}$).put($AST_W(\hat{A})$, `newNode`)

**Example 7** *Consider the ADB update example discussed in Section 3.2.2 adapted to the use of an Index Node. The index node maintains the mapping between the state variables, A and B, to the DFG nodes containing the ASTs of updates 'A=A+1' and 'B=A' respectively. When an update 'A=A+2' is applied, the DFG nodes containing the AST over its read dependencies can be directly accessded through Index node. After connecting the new node to its dependency nodes, the function $UPDATE-INDEX_I$ updates the index node's mapping and points the key A to the the update AST's node. The transformation is shown in the Figure 3.6.*

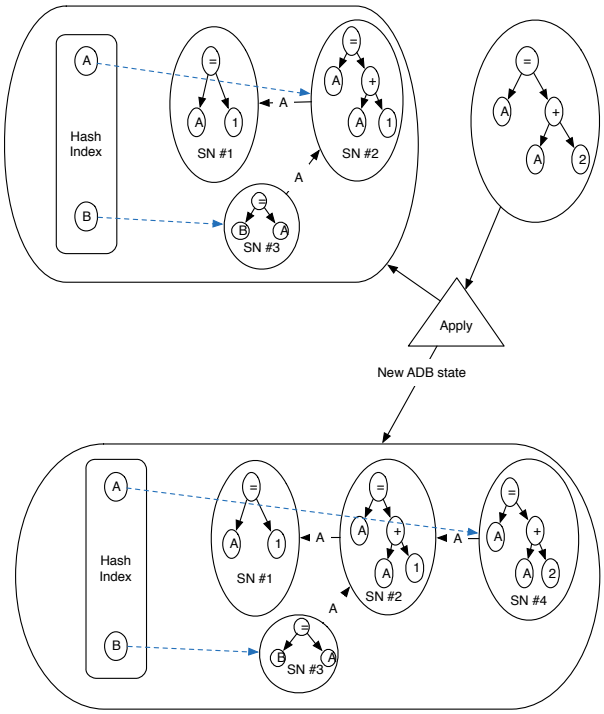



Figure 3.6: An ADB's DFG transformation in the presence of index node.

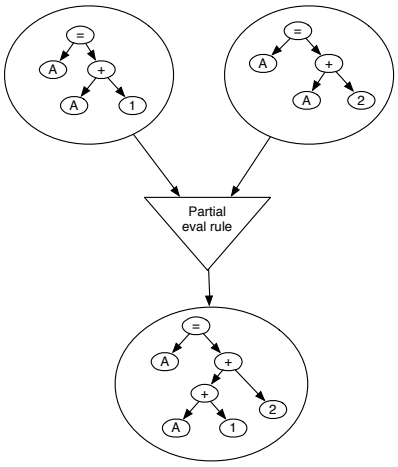



Figure 3.7: The Partial evaluation triggered rule.

### 3.3.2 Triggered AST rewrite rules: Faster Evaluation

AST rewrite rules give developers full control over how update evaluation should occur. AST rewrite rules that suite workload specific characteristics can be framed to make the evaluation of ASTs efficient. A rewrite rule consists of

1. A pattern that identifies an ADB's DFG subgraph.

2. A procedure or declarative specification that rebuilds the subgraph.

**Example 8** *An example of a rewrite rule is shown in Figure 3.7. The figure illustrates a partial evaluation AST rewrite rule that merges two ASTs that satisfy the commutative property, and stores them in a partially evaluated form.*

AST databases let developers to specify where and when the AST rewrite rules should be applied on the database state. AST rewrite rules can be applied at the time when update is performed, when the client reads from the ADB infrastructure, when the infrastructure is idle. Moreover evaluation of the ASTs can be partitioned at different times. This decision can be based upon workload characteristics or server load.

**Example 9** *The example of evaluation partition is shown in Figure 3.8. First the partial evaluation rewrite rule is applied at the time when update is performed, then the second partial evaluation rewrite rule is applied when the system is idle, and the third partial evaluation rewrite rule is applied when a client reads from the database infrastructure.*

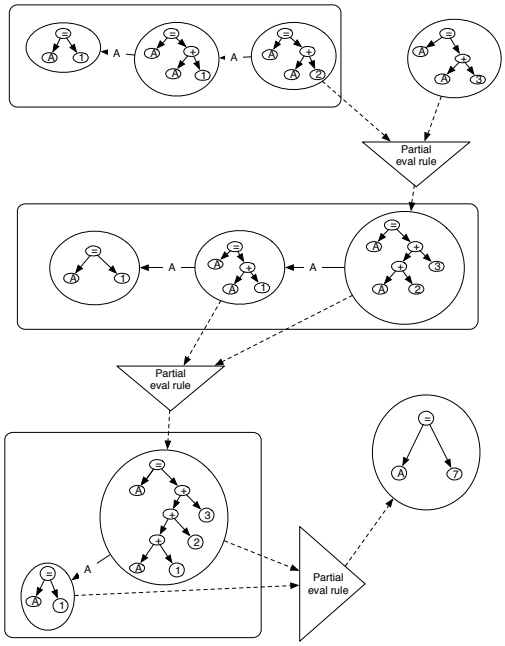



Figure 3.8: The figure illustrates the partition of ASTs evaluation.

# Chapter 4

# Laasie: An AST Database

Laasie is a document store, Abstract Syntax Tree database, specialized for hierarichal structured data. Laasie, like a generic AST database, stores a sequence of updates performed over data components. Laasie is amenable to post-hoc specialization for an application's data characteristics and access patterns. Laasie, specifically, is specialized for an ADB's primary operations:

1. Updating a reified state by appending new ASTs to the sequence of ASTs stored in its Data Flow Graph (DFG).

2. Accessing data by a varient of static program slicing [46].

3. Using pattern-matching over the DFG's structure to identify and apply potential AST rewrites.

Laasie presently includes support for a simple hierarchical data manipulation language based on Monad Algebra [31, 10] known as $Bar_{QL}$ [30]. However, Laasie allows interchangeable support for different underlying languages.

The chapter describes the semantics of $Bar_{QL}$ language, the structure of the Laasie infrastructure and focusses on the enhancements made in Laasie infrastructure over the general AST databases (discussed in previous chapter) to make it specialized for a hierarichal data model. The chapter starts by describing the hierarichal data model and the semantics of the $Bar_{QL}$ language. The rest of this chapter is dedicated to the structure of the Laasie infrastructure and explaining how it performs an AST database's primary operations: read, write, and evaluation.

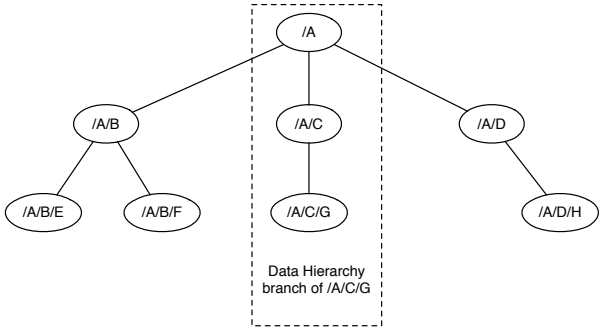



Figure 4.1: The figure shows the sample hierarichal data model. The hierarichal tree consists of three levels. The data hierarchy branch of key '/A/C/G' is highlighted in the hierarichal tree.

## 4.1 Hierarichal Data Model

A hierarichal data model is a data model in which the data is organized into a tree-like structure. This structure allows representing information using parent-child relationships: each parent can have many children, but each child has only one parent (also known as 1-to-many-relationship). In the Laasie ecosystem, individual state variables (or keys) are represented by their full path specifications or URLs. A full path specification points to a value by following the tree hierarchy, in which path components separated by the delimiting character '/' represent each tree level.

**Example 10** *A sample hierarichal data model is shown in Figure 4.1. The hierarichal tree consists of three levels with each level consisting the full path specifications (or URLs) of the keys at the respective level. Different levels of hierarichal data are referred to as the levels of the data hierarchy. For example, the key '/A/C' lies at the second level of the data hierarchy, whereas the key '/A/C/G' lies at the third level of the data hierarchy.*

*The individual branches (the paths from the root node to different leaf nodes) of the hierarichal tree are referred to as the data hierarchy branch of the key at the last level of the branch. The data hierarchy branch of the key '/A/C/G' is highlighted in the Figure 4.1. The branch consists of three levels with '/A' in its first level, '/A/C' in its second level, and '/A/C/G' in its third level.*

$$
\begin{array}{llllll}
c & \in & \mathit{Constant} :\rightarrow p & k & \in & \mathit{Key} \\
p & \in & \mathit{Primitive} & Q & \in & \mathit{Query} : \tau \rightarrow \tau \\
v & \in & \mathit{Value} & \tau & \in & \mathit{Type} : p \mid \{k_i \rightarrow \tau_i\} \mid \mathit{null} \\
\theta & \in & \mathit{BinaryOp} & & &
\end{array}
$$

$$
\begin{array}{rcl}
Q & := & Q.k \mid Q \Leftarrow Q \mid \textbf{map } Q \textbf{ using } Q \mid Q \textbf{ op}_{[\theta]} Q \\
 & \mid & \textbf{agg}_{[\theta]}(Q) \mid \textbf{agg}_{[\Leftarrow]}(Q) \mid \textbf{filter } Q \textbf{ using } Q \\
 & \mid & \textbf{if } Q \textbf{ then } Q \textbf{ else } Q \mid Q \circ Q \mid \textbf{c} \mid \textbf{null} \mid \emptyset
\end{array}
$$

Figure 4.2: Domains and grammar for $Bar_{QL}$.

.

## 4.2 The $Bar_{QL}$ Language

$Bar_{QL}$ [30] is a functional update query language based upon the Monad Algebra [31, 10]. Laasie's clients represent updates to application state as $Bar_{QL}$ queries. Update $Bar_{QL}$ queries consists of computations. In Laasie these are not evaluated but appended to the sequence of previous update ASTs stored in the infrastructure. The consequence of this is a simplified semantics for out-of-order appends and encoding of modifications (updates or deletes) as increments (i.e. deltas) rather than fixed writes (e.g. var=3). In short, $Bar_{QL}$ allows the operational semantics of updates to be managed functionally. $Bar_{QL}$ is intended to be simple, expressive, and amenable to program analysis. $Bar_{QL}$ is intentionally limited to operations with the linear computation complexity in the size of the input data; neither the pair-with nor cross product operations are included. Cross-products can be transmitted to clients more efficiently in their factorized form, and clients are expected to be capable of computing cross products locally.

The domains and grammar of $Bar_{QL}$ are shown in Figure 4.3. In the $Bar_{QL}$ ecosystem, update/modify/insert queries are divided into three components. The first component signifies the $Bar_{QL}$ token which classifies the query as the update query. The second component signifies of the full path specification of the application key which needs to be updated. The third component represents the computation that needs to be performed.

**Example 11** *The $Bar_{QL}$ query 'w /A:=3', instructs to write primitive value 3 to application key '/A'. On the other hand, the read queries consists of $Bar_{QL}$ token which classifies it as read request, and the full path specification of the application*

*key desired to be read. For example, the $Bar_{QL}$ query 'r /A' instructs Laasie to return the updates performed on application key '/A'.*

The computations in a $Bar_{QL}$ query consists of either $Bar_{QL}$ functions or values. The values in $Bar_{QL}$ function can be null or collections. Note that collections is the set of mappings. At the base, a singleton can be defined as a collection where all keys except one map to null value. By convention, when referring to collections we will implicitly assume the presence of this mapping for all keys that are not explicitly specified in the functions themselves.

In $Bar_{QL}$, functions are monad structures that represent computation. The rules for Primitive Constant, Null and Empty Set all define operations that take an input value and produce a constant value regardless of input. The rule Primitive Constant produces the primitive constant, the Null produces the null value, and the rule EmptySet produces an empty set. Empty Set is defined as the collection that is a total mapping, where all keys map to null value.

The Identity operation passes through the input value unchanged. Subscripting and Singleton are standard operations. In comparison to Monad Algebra, these operations correspond to not only the singleton operation over sets, but also the tuple constructor and projection operations. Because collection elements are identified by keys, we can reference specific elements of the collection in much the same way as projection on a tuple.

The merge function combines two (or more) sets, replacing undefined entries in one collection (keys for which a collection maps to null) with their values from the other collection:

$(\{\text{A:=1}\} \Leftarrow \{\text{B:=2}\})(\text{null}) = \{\text{A} \rightarrow 1, \text{B} \rightarrow 2\}$

If a key is defined in both collections, the right input takes precedence:

$(\{\text{A:=1}\} \Leftarrow \{\text{A:=2}\})(\text{null}) = \{\text{A} \rightarrow 2\}$

The merge operator can be combined with singleton and identity to define updates to collections:

$(\text{id} \Leftarrow \{\text{A:=3}\})(\{\text{A} \rightarrow 1, \text{B} \rightarrow 2\}) = \{\text{A} \rightarrow 3, \text{B} \rightarrow 2\}$

Subscripting can be combined with merge, singleton, and identity to define point modifications to collections:

(id $\Leftarrow$ {A:=(id.A $\Leftarrow$ {B:=2})}) ({A $\rightarrow$ {C $\rightarrow$ 1}}) = {A $\rightarrow$ {B $\rightarrow$ 2, C $\rightarrow$ 1}}

If a key is defined in both collections, the right input takes precedence:

({A:=1} $\Leftarrow$ {A:=2})(null) = {A $\rightarrow$ 2}

The merge operator can be combined with singleton and identity to define updates to collections:

(id $\Leftarrow$ {A:=3})({A $\rightarrow$ 1, B $\rightarrow$ 2}) = {A $\rightarrow$ 3, B $\rightarrow$ 2}

Subscripting can be combined with merge, singleton, and identity to define point modifications to collections:

(id $\Leftarrow$ {A:=(id.A $\Leftarrow$ {B:=2})}) ({A $\rightarrow$ {C $\rightarrow$ 1}}) = {A $\rightarrow$ {B $\rightarrow$ 2, C $\rightarrow$ 1}}

Primitive binary operators are defined monadically with operation PrimBinOp, and include basic arithmetic, comparisons, and boolean operations. These operations can be combined with identity, singleton, and merge to define updates. For example, to increment A by 1, we write:

{id $\Leftarrow$ {A:=id.A+1}}({A $\rightarrow$ 2}) = {A $\rightarrow$ 3}

$Bar_{QL}$ provides constructs for mapping, flattening and aggregation. The map operation is analogous to its definition in Monad Algebra, save that key names are preserved. The Map operation applies the computation to all the child of the input collection. For example, to increment all children of the root by 1, the map operation can be written as follows;

(map id using (id+1))({A $\rightarrow$ 1, B $\rightarrow$ 2}) = {A $\rightarrow$ 2, B $\rightarrow$ 3}

To increment the child C of each component of the root by 1, the map operation can be written as follows:

(map id using (id $\leftarrow$ {C:=id.C+1})) ({A $\rightarrow$ {C $\rightarrow$ 1}, B $\rightarrow$ { C $\rightarrow$ 2, D $\rightarrow$ 1}}) = {A $\rightarrow$ {C $\rightarrow$ 2}, B $\rightarrow$ {C $\rightarrow$ 3, D $\rightarrow$ 1}}

The Flatten operation is also similar to the one in Monad Algebra, expect that it is based on $\Leftarrow$, rather than $\cup$ as in Monad Algebra.

To overwrite the child C of each component of the root by 5, the flatten operation can be written as follows:

(flatten id using $\{$C:=2$\}$)($\{$A $\rightarrow$ $\{$C $\rightarrow$ 1$\}$, B $\rightarrow$ $\{$ C $\rightarrow$ 3, D $\rightarrow$ 1$\}\}$) = $\{$A $\rightarrow$ $\{$C $\rightarrow$ 2$\}$, B $\rightarrow$ $\{$C $\rightarrow$ 2, D $\rightarrow$ 1$\}\}$

The Filter operation can be used for eliminating/deleting the application components which are no longer needed or of no interest. To delete the collection from C whose child C's value is not 1, the filter operation can be written as:

(filter id using $\{$C:=1$\}$)($\{$A $\rightarrow$ $\{$C $\rightarrow$ 1$\}$, B $\rightarrow$ $\{$ C $\rightarrow$ 2, D $\rightarrow$ 1$\}\}$) = $\{$A $\rightarrow$ $\{$C $\rightarrow$ 1$\}$

Finally, $Bar_{QL}$ also supports Conditionals and Composition of queries. The rules for these reductions are standard.

### 4.2.1 AST Analysis

Recall that the ASTs stored in an AST database (as defined in chapter 3) represent updates performed over the application state variables. The ASTs are generated by analysing the syntax of the updates represented in data manipulation language, built over ADB infrastructure. In the Laasie ecosystem, $Bar_{QL}$ is the data manipulation language on which, similar to any programming language, program analysis techniques can be applied.

The $Bar_{QL}$ ASTs are amenable to program analysis. This allows developers to fetch the following metadata information from the ASTs.

1. AST write set: The write set of an AST includes the keys which are updated by the AST. Given the AST $\hat{A}$, the function $AST_W$ generates the URLs of the keys in AST write set.

2. AST read set: The read set of an AST includes the keys on which AST depends i.e. the keys whose value is required to evaluate the AST. Given the AST $\hat{A}$, the function $AST_R$ generates the URLs of the keys in AST read set.

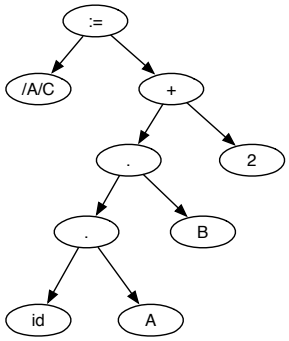



Figure 4.3: The AST of the computation /A/C:=id.A.B+2

3. Data Hierarchy branch: Given the URLs of the keys in AST read and write set, the function $AST_{DH}$ returns the data hierarchy branch of respective keys. The number of levels in the data hierarchy branch is equal variable's data hierarchy tree level.

The functions $AST_W$, $AST_R$, and $AST_{DH}$ are called metadata extraction functions. The implementation of metadata extraction functions is connected to the semantics of the $Bar_{QL}$ language.

**Example 12** *Consider the AST of the update '/A/C:=id.A.B+2' as shown in Figure 4.4. Since the update has been performed over the key '/A/C', the function $AST_W$ will return '/A/C' in write set. In order to evaluate the AST, the value of key '/A/B' is required and the function $AST_R$ will return the key '/A/B' in read set. Given the URL '/A/C', the function $AST_{DH}$ generates the data hierarchy branch consisting of two levels, with first level containing the key '/A' and the second level containing the key '/A/C'.*

## 4.3 Laasie Data Flow Graph

Laasie stores sequences of update ASTs in its Data Flow Graph (DFG). Because Laasie is specifically developed for a hierarichal data model, its DFG is different than that of a generic AST database (discussed in Chapter 3). The main difference between the two is the index structure built over the data flow graph. Recall that a hash index is used over the DFG of a generic AST database when fetching sequences of updates required to evaluate the reified value of individual data components. As opposed to a generic AST database, which deals with disjoint state

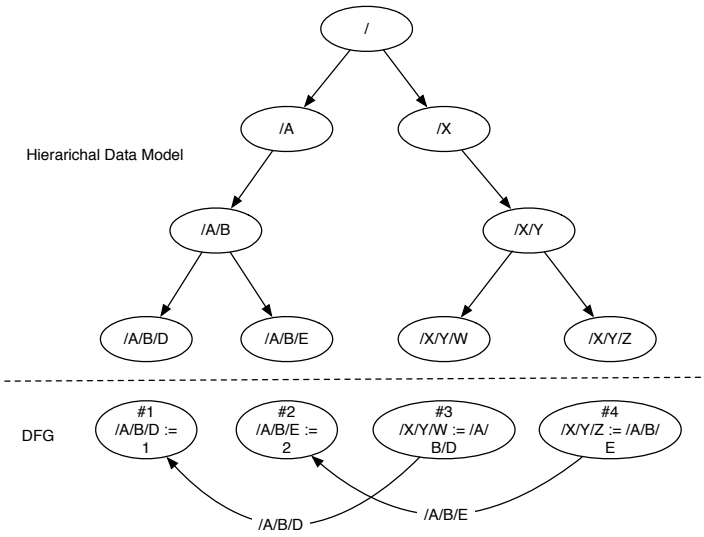



Figure 4.4: The DFG and the hierarichal model of the keys

variables, Laasie is specifically built for a hierarichal data model where state variables (or keys) may overlap (for example the key '/X' is a superset of key'/X/Y'). The Laasie infrastructure should be able to support queries involving user defined patterns over keys (for example, find all the updates performed over keys starting with '/X') in addition to individual keys.

Consider the data flow graph and the hierarichal model shown in Figure 4.5. The data flow graph stores the AST of four updates to the database's state. DFG nodes are connected to other nodes which store ASTs of the dependent keys. The figure also shows the hierarichal model of the keys whose updates are stored in the DFG. Consider a query which finds all the updates performed over keys starting with '/X/Y'. The keys which satisfy the query pattern are '/X/Y/W' and '/X/Y/Z'. Collecting updates performed over key '/X/Y' is equivalent to collecting the updates performed over keys '/X/Y/W' and '/X/Y/Z'. Similarly, collecting updates performed over key '/X' is equivalent to collecting updates over data component '/X/Y'. *This can be generalized to say that collecting update performed over key, lying at the $n^{th}$ level of data hierarichal tree, is equivalent to collecting updates performed over its child keys, lying at $(n+1)^{th}$ level of data hierarichal tree.*

This observation suggests that queries involving patterns over keys can be supported efficiently with a tree-like index structure over DFG nodes storing update ASTs. The index tree, similar to the data hierarichal tree, can be constructed over

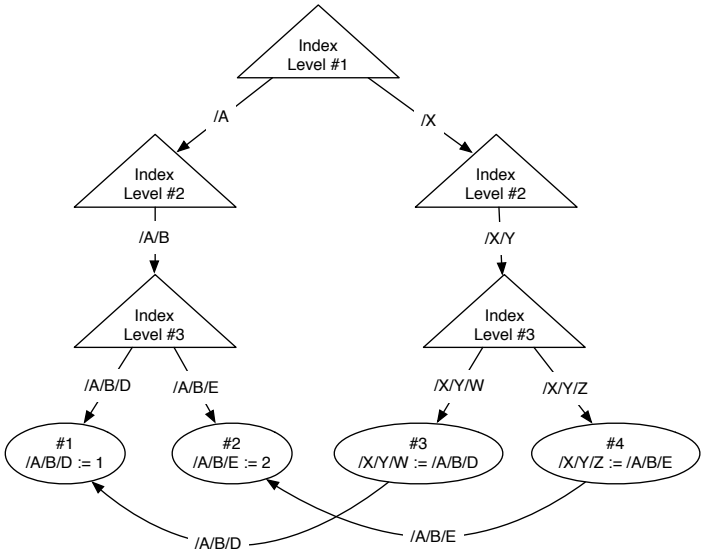



Figure 4.5: The Index tree, similar to hierarichal tree, built over DFG.

the DFG nodes containing update ASTs as shown in Figure 4.6. An important point to note here is that the depth (i.e. number of levels) of the index tree is equal to the highest level of the data hierarichal tree on which the updates are performed. This constraint is always enforced on the index tree, since it ensures that the full data hierarchy branches of the data components (on which the updates are performed) are present in the index tree.

The Laasie DFG denoted as LDFG=(V, E, $I_R$) with

1. V is the set of vertices (or nodes) in Laasie's DFG. Nodes in the data flow graph can be categorized as index nodes and storage nodes. Index nodes are the part of the index tree, while storage nodes store the AST of the updates performed over application state variables. As opposed to nodes in a generic ADB's DFG which maintains an adjacency list, nodes in Laasie's DFG maintains a mapping from the node's outgoing edge labels to the DFG nodes pointed by the edges. This makes the traversal over the index tree efficient. The mappings maintained in each node are represented by the function MAP(node), where node is the DFG node.

2. E is a set of node ordered pairs. An edge e = $(x, y)^z$ is considered to be a directed edge from storage node x to storage node y. The AST stored in node x is said to be dependent upon the AST stored in node y. The dependency

attribute between the two ASTs is denoted by z.

3. $I_R$ is the root index node i.e. index node at the first level of the index tree.

### 4.3.1 Read in Laasie

Recall that the hash index, built a over generic AST database's DFG, maps application state variables to the nodes storing the latest update AST. Once the latest update is found, the updates over its read dependencies can be found by performing breadth first search from the storage node containing latest update AST. Recall that while applying breach first search, actions have to be taken to eliminate duplicate nodes encountered and to maintain the original order of the updates (by using a priority queue).

The Laasie infrastructure traverses the tree index to find the source node from which the breadth first search can be applied to collect the descendant nodes. Given a key to be read, the respective data hierarchy branch is traversed in the index tree. The function $Read_I$ is used to traverse the index tree. The function uses metadata extraction function $AST_{DH}$ to find the data hierarchy branch of the data component. In a nutshell, the function $Read_I$ follows the index node's outgoing edges with label of the data component present at the respective level of the data hierarchy branch.

**Algorithm 7** $Read_I$(LDFG(V, E, $I_R$), $URL_A$)

**Require:** `LDFG(V, E, `$I_R$`)`, The current ADB's DFG.
**Require:** $URLA$, The URL of the data component, A.
**Ensure:** `rootNode`, The root node containing the latest operation over data component A.

```
rootNode = I_R
dataHierarchy ← AST_DH(URL_A)
for U ∈ dataHierarchy do
    nextNode = MAP(rootNode).get(U)
    if nextNode ≠ null then
        rootNode = nextNode
    end if
end for
```

Once the source node is returned by function $Read_I$, breadth first search can be applied to collect all the descendant nodes. The function $Read_D$ is used to

**Algorithm 8** $Read_D$(LDFG(V, E, $I_R$), sourceNode)

**Require:** `LDFG(V, E, I_R)`, An ADB's DFG.
**Require:** `sourceNode`, The source node from which BFS has to be applied.
**Ensure:** `descendants`, The set of descendants of the selected root nodes.

```
Create Priority Queue Q
Q ← Enqueue(sourceNode)
descendants = ∅
descendants = descendants ∪ sourceNode
while Q ≠ Empty do
  node ← Dequeue(Q)
  for mapping ∈ MAP(node) do
    dependentNode ← mapping.getNode()
    if dependentNode not in Q then
      if TYPE(dependentNode) ≠ INDEX then
        descendants = descendants ∪ dependentNode
      end if
      Enqueue dependentNode onto Q
    end if
  end for
end while
```

collect the descendant nodes. $Read_D$ applies breadth first search over the source node and eliminates duplicate nodes encountered during traversal. The function also eliminates index nodes while traversing and keeps only storage nodes in the set of descendant nodes since index nodes do not store any update ASTs.

Note that, depending upon the state variable, the function $Read_I$ can return the storage or index node. The common strategy of applying breadth first search remains same irrespective of the type of node returned by function $READ_I$. However, if the node returned is an index node, the function $READ_D$ has to make sure to eliminate the index nodes (encountered during traversal) from the set of descendant nodes, since they do not store the update ASTs.

### 4.3.2 Update in Laasie

Recall that in a generic AST database, new update gets appended to the stored sequence of updates by creating new DFG nodes and connecting to other nodes storing its dependent ASTs, which are fetched using the hash index. Appropriate changes are made to the hash index to point to the new update. The update process in the Laasie infrastructure follows the similar process. The only difference is that the nodes containing the dependent ASTs are fetched by traversing the index tree and appropriate changes have to made to the index tree point to the new update.

Given an update AST $\hat{A}$, the function $Update_V$ is used to create a new DFG storage node which stores the update's AST. Once the new node is created, the function $Update_E$ connects it to the existing DFG nodes which contains the latest updates on new update's dependent data components. Recall that the AST read dependencies can be extracted using metadata extraction function $AST_R$. The function $Update_E$ uses the approach similar to $Read_I$ to traverse the index tree for fetching the dependent DFG nodes.

The Laasie infrastructure enforces a constraint on the index tree which asserts that the depth of the index tree is always equal to the highest level of the data hierarchy tree on which the updates are performed. This constraint ensures that full data hierarchy branches of all keys (on which the updates are performed) are present in the index tree. Thus the modification of the index tree depends upon the data hierarchy level of the data component being updated. If the data hierarchy level of the component is greater than the depth of the index tree, new levels are inserted. The number of levels to be inserted is equal to the difference between the data hierarchy level of the updated key and the depth of the tree. If the data hierarchy level of the updated key is less than the depth of the index tree, no new levels need to be inserted but the existing index node changes to point to the new

**Algorithm 9** $Update_E$(LDFG(V, E, $I_R$), $\hat{A}$)

**Require:** `LDFG(V, E, `$I_R$`)`, The current ADB's DFG.
**Require:** $\hat{A}$, The AST of the new update.
**Ensure:** $\hat{L}$`DFG(`$\hat{V}$`, `$\hat{E}$`, `$I_R$`)`, The modified DFG containing the new node and its dependency edges

```
newNode ← Update_V(Â)
V̂ = V ∪ newNode
readDepSet ← AST_R(Â)
for readDep ∈ readDepSet do
    rootNode ← I_INDEX(LDFG(V, E, I_R), readDep)
    Ê = E ∪ E(newNode, rootNode)^readDep
end for
```

**Algorithm 10** $Update_I$(LDFG($\hat{V}$, $\hat{E}$, $I_R$), $\hat{A}$)

**Require:** `DFG(`$\hat{V}$`, `$\hat{E}$`, `$I_R$`)`, The modified DFG.
**Require:** $\hat{A}$, The AST of the new update.
**Ensure:** `DFG(`$\hat{V}$`, `$\hat{E}$`, `$\hat{}I_R$`)`, The DFG with modified index structure.

```
newNode ← Update_V(Â)
previousNode = I_R
currentNode = I_R
URL_W ← AST_W(Â)
dataHiearchy ← AST_DH(URL_W)
for component ∈ dataHiearchy do
    currentNode = MAP(previousNode).get(component)
    if Type(currentNode) ≠ INDEX then
        indexNode ← Node(INDEX)
        MAP(previousNode).put(component, indexNode)                    #
        Add the new entry to the node mapping. It is equivalent of adding the new edge to DFG.
        currentNode = indexNode
    end if
    previousNode = currentNode
end for
MAP(currentNode).put(URL_W, newNode)
^I_R = I_R
```

update as it is now the latest update performed.

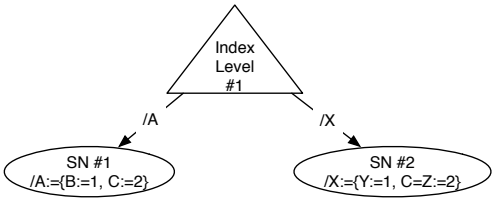
The function $Update_I$ is used to modify the index tree in case of an update. the function traverses the data hierarchy branch of the updated data component in the index tree. If the full data hierarchy branch does not exist, it creates the level of the index node at the missing levels. If the full data hierarchy branch is available, it points the last level index node to the node containing the latest update AST.

### 4.3.3 Examples

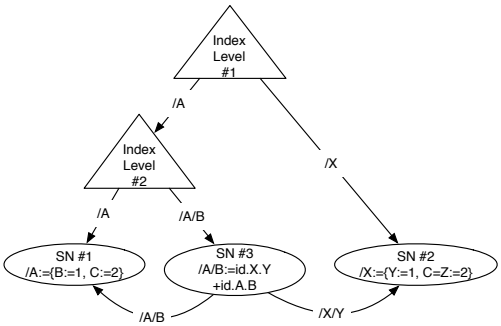
This section illustrates the read and update processes with the examples.

**Update Example** Consider the Laasie DFG as shown in Figure 4.7a. The DFG consists of two storage nodes which initialize the collections '/A' and '/X'. Since the highest data hierarchy level on which the updates are performed is one (i.e. '/A' and '/X'), the DFG consists of only one level of the index node. Figure 4.7b shows the modified DFG after the update has been performed over the key '/A/B'. The function $Update_V$ creates a new node to store the update AST and the function $Update_E$ connects it to its dependent nodes. Note that the update is been performed on the key which lies at the second level of the data hierarchy and the current depth of the index tree is one. Thus, one more level of the index node is is inserted (Index level #2) as shown in figure. Figure 4.7c shows the modified DFG after an update overwrites key '/A/C'. The update is performed on a key which lies at the second level of the data hierarchy, which is the current depth of the index tree. Thus no new levels are inserted in the index tree, but an outgoing edge is created from second level index node to the new node containing the update AST. Figure 4.7d shows the modified DFG after the update has been performed over the key '/X/Y'. The update overwrites the key with the primitive constant 10. Similar to the previous case, no new levels of are added in the index tree, but new index nodes are added in the second level of the index tree as shown in the figure.

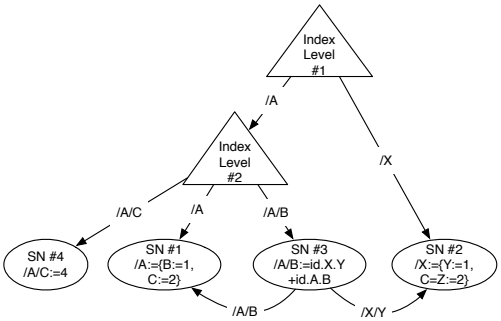
**Read Example** Consider the Laasie DFG as shown in Figure 4.7d. Suppose the user intents to collect the sequence of operations performed over key '/A/B'. The function $Read_I$ iterates over the index tree to find the storage node 2 (SN #2 in Figure 4.7d) as the source node. The function $Read_D$ collects all the descendant nodes which stores the dependent ASTs. The function returns the storage nodes #1, #2, and #3. Next consider a user request for all the updates over the key '/A'. In this case the function $Read_I$ returns the index node at level #2 as the source node.



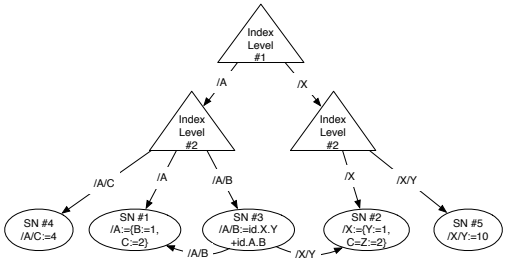

(a) Initial Laasie DFG




(b) Laasie DFG after update '/A/B:=id.A.B+id.X.Y'

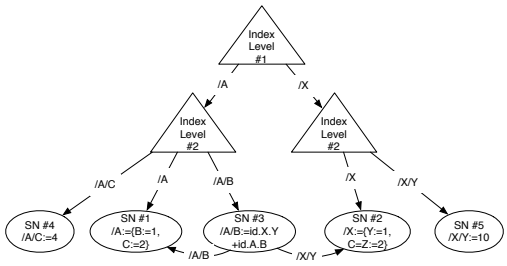



(c) Laasie DFG after update '/A/C:=4'

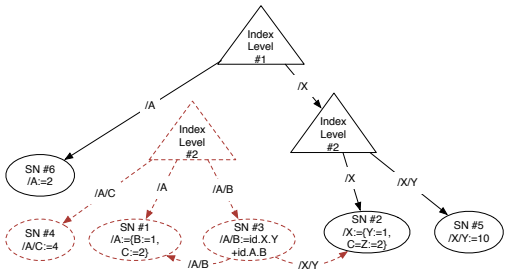



(d) Laasie DFG after update '/X/Y:=10'

Figure 4.6: Laasie Update Example




(a) Initial Laasie DFG




(b) Laasie DFG after update '/A:=2'

Figure 4.7: The figure illustrates the Laasie DFG's self cleaning capability.

Given the source node, the function $Read_D$ collect all the descendant nodes of it. The index nodes encountered during the traversal will be eliminated returning the storage nodes #1, #2, #3 and #4 as the descendant nodes.

### 4.3.4 Self Cleaning DFG

Laasie stores sequences of updates, suggesting that the space requirements to store these updates can be huge in case of large number of updates over keys. In such a case, the infrastructure developer needs to detect and delete the unwanted nodes on timely basis. The main advantage of using the tree-like data structure for storing updates is that it automatically detects the unwanted nodes. The Laasie infrastructure is developed in Java, and deletion of the unwanted nodes can be left to the JVM's garbage collector.

**Example 13** *Consider the Laasie DFG shown in the Figure 4.8a. The Laasie DFG shown consists of two levels of index nodes since the highest level on which the updates are performed is two ('/A/B', '/A/C', and '/X/Y'). Consider an update*

*that overwrites the whole collection '/A' with primitive constant 2. Since the update has been performed over the data component which lies at the first level of data hierarchy, the update function Laasie − Update$_I$ points the first level index node to the new node containing the update. This leaves the index node at level #2 and storage nodes #1, #3, and #4 unreferenced (as shown in figure 4.8b) which can later be collected JVM garbage collector.*

### 4.3.5 Evaluation of ASTs

The update ASTs collected by functions $READ_I$ and $READ_D$ need to be evaluated to obtain a key's reified value. Like any other AST database, Laasie provides user with full control over where and when this evaluation happens. The evaluation can happen server side or can be offloaded to clients. The user can also decide when to perform the evaluation. It can be performed when update has been applied to database, when the user reads from the database, when the infrastructure is idle (i.e. not processing any client requests). This decision can be based on factors like server load, network bandwidth, workload characteristics and requirements, or client computation capacity.

#### Triggered AST rewrites

As an AST database, Laasie supports the triggered AST rewrites. An AST rewrite rule consists of:

1. A pattern that identifies an data flow graph subgraph.
2. A procedure or declarative specification that rebuilds the subgraph.

AST rewrite rules can be specified over the Laasie infrastructure which better suites the workload characteristics and requirements. Although check is made for the specified rewrite patterns on every application update by default, the Laasie infrastructure provides user the full control over when such checks should be triggered.

**Partial Evaluation** One such example of a triggered AST rewrite rule is partial evaluation, which combines two composable update ASTs and stores them in partially evaluated form. For example, consider the AST of an update 'A:=A+1'. If the Laasie client increments the value of key A by 2, the two ASTs, which are

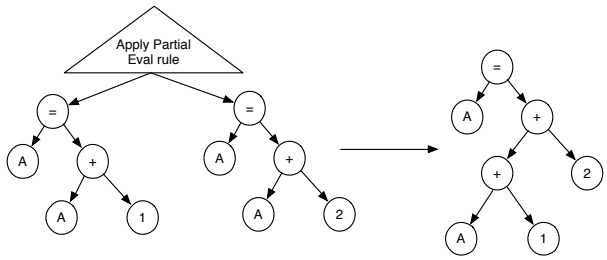



Figure 4.8: Since the two computations are composable in nature, they will be stored in the partially evaluated form when applied to partial evaluation rewrite rule.

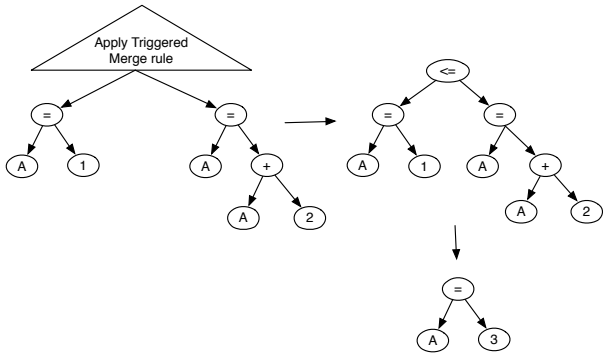



Figure 4.9: The two computations update the same application key. They will be merged together when merged rewrite rule is triggered. The resulting merged AST can be evaluated further.

composable in nature, will be stored in the partially evaluated form subject to partial evaluation rewrite rule. The process is illustrated in Figure 4.9. Because of the partial evaluation rewrite rule, the underlying DFG structure will store only the result of the AST transformation instead of storing the two ASTs separately. This makes the underlying infrastructure more space efficient and gives developer more fine grained control over the infrastructure.

**Triggered Merge** Another example of the triggered AST rewrite is triggered merge which merges the new value of the application key updated with its previous value and store the result of the merge operation. Consider the DFG containing the AST of the update 'A:=1'. If the Laasie client increments the value of A by 1, the triggered merge rule merges new $Bar_{QL}$ update function AST with

previous existing AST resulting in the new merged AST as shown in Figure 4.10. The resulting AST can be evaluated further resulting in the final AST of result 'A:=3'. Similar to the partial evaluation rewrite rule, the triggered merge rewrite rule makes the underlying infrastructure more space efficient and gives developer more fine grained control over database insfrastrucutre.

## 4.4 Collaborative Web Applications: Motivating use case for Laasie

Applications like Google Docs [22], Office 365 [15], or Dropbox [18] signify a shift from traditional desktop applications towards a more collaborative environment where multiple users can simultaneously view and edit the same document(s) in near real time. This class of collaborative applications are based upon client/server model, where application core functionality lies at the client side and server is used to persist the shared application state. The server's role is to relay edits/updates made (to shared application state) by any user to all other connected users. Such applications present a challenge, because the state machine replication [39] capabilities that they employ require a combination of persistence and publish/subscribe capabilities that to the best of my knowledge is not provided by existing data management systems [4].

Data management in this domain is hindered by by challenges such as:

1. **Non Uniform State**: A single web application may need to store and access many different classes of data (*e.g.*, key/value pairs, searchable indexes, GIS or graph data, *etc*...).

2. **Evolving requirements**. Centralized hosting allows web application developers to rapidly deploy new features and functionality, resulting in frequent, live changes to the application's workload, schema, and data management needs.

3. **Bursty, unpredictable growth**. Termed the slashdot effect, a prominent exposure of the web application (*e.g.*, on a news aggregator site), can result in a sudden, unexpected dramatic growth in traffic as users begin to refer their friends.

These challenges, combined with general performance and scalability requirements, result in developers frequently resorting to hybrid infrastructures cobbled together from multiple distinct data management systems. Collaborative

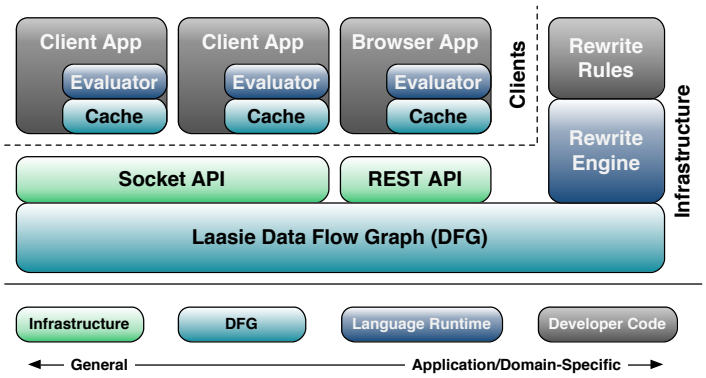



Figure 4.10: Laasie's system architecture.

applications have received considerable interest, both from industry [45], and academia [35, 37]. At present, most efforts to support collaborative applications are client-side, providing a client-driven persistence and notification service [45], techniques for automatically extracting shared state from single-site application [26], or generic collaborative widgets [45]. Infrastructure efforts to date have primarily focused on traditional one-size-fits-all engineering efforts [45], or on accessing specific properties such as reliability [35].

The same variability that makes collaborative applications hard for any one data management system to support makes Laasie an ideal fit. Laasie's AST triggered rewrite rules allow extremely fine grained control over the exact way the application state is stored. Simultaneously, a single, consistent interface allows the front-end to be developed independently of any infrastructure optimizations. This allows for rapid prototyping and early experimentation to obtain detailed workload characteristics before the infrastructure is specialized.

A high level view of Laasie's architecture is shown in Figure 4.11. Typical clients interact with Laasie's server through a thin library component. This component maintains a replica of one or more slices of core DFG. When reified values are needed, this library layer is capable of evaluating the appropriate cached slice. To maintain the reified state, client sends $Bar_{QL}$ queries to a Laasie server, where they are parsed and composed with the existing sequence of ASTs and subjected to rewrite rules. The server-side reqrite rules may evaluate the update, or leave it intact as a delta (or both). If appropriate, the updated state is then reflected back to the client during the nect update.

We expect that Laasie will be used alongside a set of library datatypes built on top of the thin caching library, using Laasie as a replication layer. When such

datatypes are used, datatype-specific rewrite rules (or templates for rewrite rules) may also be possible. This approach completely hides the underlying data management system from developers, while retaining both generality and Laasie's ability to specialize.

Client caches stay up-to-date using a form of State Machine Replication [39]. As discussed above, read requests include a summary of AST nodes already cached at the client. When slicing, labeled nodes are replaced with placeholders, allowing the server to send only DFG nodes inserted since the client was last updated, in effect replaying these updates against the client' state. A similar approach that uses push-based semantics (e.g., as in LiveObjects [35]) is also possible, but beyond the scope of the thesis.

# Chapter 5

# Benchmark and Experiments

The chapter discusses the evaluation of Laasie and demonstrate the feasibility of AST databases. We evaluated our document store database, Laasie, against several other commercial and open source data management systems. The experiments uses two benchmark suites - Yahoo Cloud Benchmark (YCSB) and the Collaborative application benchmark. The evaluation shows that on benchmark workloads not optimized for use with Laasie (YCSB), Laasie's performance was within an order of magnitude of the significantly more mature comparison points. On a benchmark designed to simulate the persistence and data replication requirements of a collaborative application, Laasie outperformed and/or showed better scaling characterstics than a commercial document store.

## 5.1 Benchmark Details

The evaluation uses two benchmark suites: Yahoo Cloud Benchmark (YCSB) and a collaborative application benchmark.

### 5.1.1 YCSB: Yahoo Cloud Benchmark

The Yahoo Cloud services benchmark is a standard benchmarking framework which facilitates the performance comparisons of the new generation of cloud serving systems. The benchmark focuses on systems that provide online read-/write access to data. The framework consists of a workload generating client and a package of standard workloads that cover interesting parts of the performance space (read-heavy workloads, write-heavy workloads, scan workloads, etc.). An

important aspect of YCSB framework is its extensibility: the workload generator makes it easy to define new workload types, and it is also straightforward to adapt the client to benchmark new data serving systems. The YCSB framework and workloads are available in open source so that developers can use it to evaluate systems, and contribute new workload packages that model interesting applications.

### YCSB Benchmark Workloads

YCSB is a Java-based benchmark that uses synthetic workloads to emulate access patterns typical to cloud services. Each workload represents a mix of read/write operations, data sizes, request distributions, and so on, and can be used to evaluate systems at one particular point in performance space. Each operation in workloads is randomly choses to be one of:

1. Insert: Insert a new record.
2. Update: Update a record by replacing the value of one field.
3. Read: Read a record, either one randomly chosen field or all fields.
4. Scan: Scan records in order, starting at a randomly chosen record key. The number of records to scan is randomly chosen.

Due to unimplemented support for scan in Bar-QL, the scan operations in YCSB are not taken into account for performance comparison of Laasie with other systems.

The YCSB benchmark consists of following six workloads:

1. Workload A: 50% reads, 50% writes.
2. Workload B: 95% reads, 5% writes.
3. Workload C: 100% reads.
4. Workload D: 95% reads, 5% inserts, with recently inserted records accessed more frequently.
5. Workload E: 95% scans, 5% inserts.
6. Workload F: 50% reads, 50% read/modify/writes.

### 5.1.2 Collaborative Application Benchmark

The Laasie group has developed a suite of benchmarks that simulate different collaborative applications to justify Laasie's usefulness for such kind of applications. The next section goes over the details of the benchmarks developed. The collaborative application benchmark consists of three real time collaborative applications: Paint, Spreadsheet and Text Editor.

**Paint** The paint workload corresponds to a collaborative paint application where several users can draw and change the pixel colors. The workload starts by initializing the empty user paramaterized rectangular grid. The rectangular grid is treated as collection that consists of a mapping of pixels (identified by x,y coordinates) and the color. The user can update a particular pixel's color. The workload also includes shape drawing operations. For testing purpose the shapes and the colors are generated randomly by Laasie workload generator.

**Spreadsheet** The spreadsheet workload corresponds to a collaborative spreadsheet application where several users can write, delete or update a sheet's cell values. The workload starts by initializing the user defined schema (i.e. a rectangular grid in the spreadsheet) with default values. The spreadsheet is treated as a collection that contains the mapping of cells (identified by x, y coordinates) and their respective values. The spreadsheet cells are capable of storing functions (with dependencies) as well. The write operations consists of updating or deleting existing cells and inserting new cells. For testing purpose the cell values and the functions are generated randomly by Laasie workload generator.

**Text Editor** The text workload corresponds to a collaborative text editor application where several users can write to a shared document. The workload starts by initializing an empty document. The document is treated as a collection that consists of several lines. The line, in turn, is also collection that consists of a list of words. The write operations consists of deleting, updating the existing lines (or words) or inserting the new lines (or words). The schema of the application continuously grows due to the higher percentage of inserts operations. For testing purpose the words are generated randomly by Laasie workload generator.

## 5.2 Experiments and Results

As comparison points, we used a relational database: MySql v14.14 [5] with both the InoDB and MEMORY engines, and a commercial document store: MongoDB v2.4.7 [13], each used without replication.

### 5.2.1 Experiment setup

All experiments were executed on a 2x16-core 1.8 GHz Opteron (32 cores total) with 128GB of RAM, two 146GB 15k RPM disks in a RAID1 configuration, and running Redhat Enterprise Linux version 6 and OpenJDK 1.7.0 45. All benchmarks were run with separate client and server binaries, both on the same machine, using each engine's native Java drivers: MongoDB Java Driver v2.10.1 and MySQL JDBC connector 5.1.17. All data management systems tested were given unrestricted access to the full hardware resources; MySQL's MEMORY engine's memory limits were set at 10GB. Unless otherwise noted, Laasie was configured to inline and evaluate (i.e., checkpoint) operations more than 1000 versions old, every 1000 operations.

### 5.2.2 YCSB Results

The YCSB benchmark evaluates database performance according to throughput and latency at varying levels of client parallelism. Before each trial, each database was reset to its initial state and populated with 100,000 records, sized 1k each. Due to unimplemented support for scans in Bar-QL, we did not obtain performance results on YCSB workload E.

Each YCSB workload was run for 100,000 operations to hide startup costs. For every YCSB workload, the throughput had been calculated against the read latency, update latency (resp, write latency) for workload A and workload B (resp, for workload F) by varying the number of threads performing operations on back-end database. The overall throughput was measured in operations per second and latencies were measured in milli-seconds. Update operations corresponds to updating the existing records in database, whereas the write (or insert) operations corresponds to inserting the new records into the database on the fly.

**Results**

Results for YCSB benchmark are shown in the figures 5.1, and 5.2. The YCSB benchmark evaluates applications for use as a persistence layer in a traditional cloud service. Although this is not the primary target use-case for Laasie, it provides the well known datapoint for evaluating its performance characterstics.

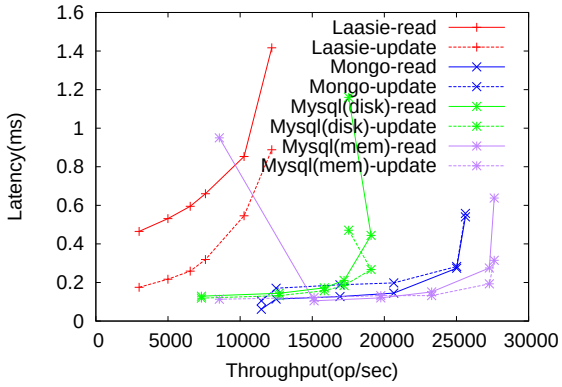
From the graphs of Workload A, B and C it can be observed that Laasie was not as performant as other commercial and stable data management systems (i.e. MySQL and MongoDB). Laasie consistently achieved performance between 10 and 20 thousand operations per second, generally no more than a factor of 2-3 slower than other data management systems we compared against. But it can be seen from the graphs that Laasie performed consistently throughout the workloads, showing the feasibility of AST databases. MongoDB's heavy optimization for read (and not insert) based workloads is evident in Workloads B and C.

For Workload D, which contains 95% read operations and 5% insert (inserting new values in database on the fly) operations, Laasie performance was better than MongoDB. Laasie achieved a performance of 17 thousand operations per second with 16 threads, as opposed to MongoDB which achieved the throughput of 10 thousand operations per second with same number of threads. This shows Laasie scales better than MongoDB for workloads with insert operations. Laasie represent inserts in the form of ASTs, with additional metadata information. Keeping this extra information does not hurts Laasie's performance, and helps it to accommodate inserts more easily than other data management systems.

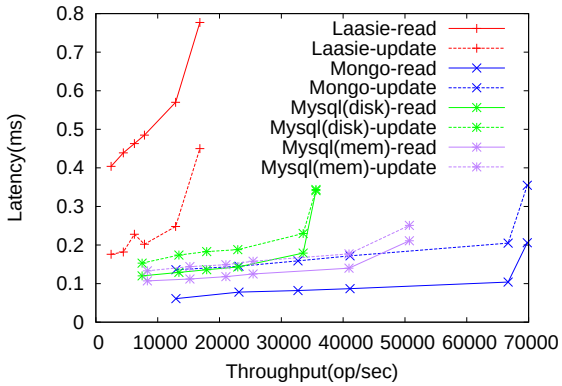
The performance bottleneck in Laasie was a consequence of coarse-grained reader/writer locking around the Directed Acyclic graph it maintains. Replacement of coarse-grained locking with concurrent graph structure data structure is expected to substantially improve the Laasie's performance, bringing it inline with other data management systems. The read/modify/write operations of YCSB workload F can be implemented in BarQL. Although YCSB models this operation as an explicit read and write, an BarQL implementation could be written directly into Laasie, reducing the access cost by one network roundtrip.

### 5.2.3 Collaborative Application Benchmark Results

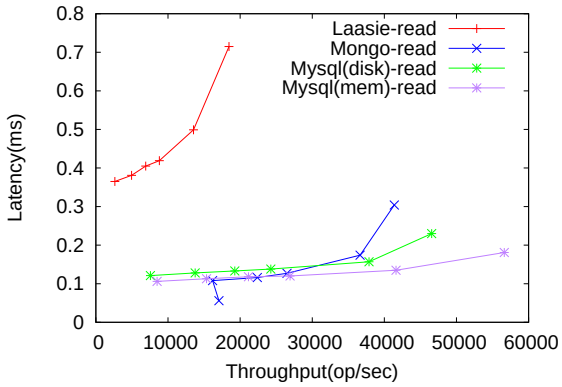
The Collaborative application benchmark is composed of synthetic workloads, which better highlight the benefits afforded by Laasie. The performance of Laasie was compared with that of MongoDB using the collaborative benchmark. Each synthetic workload creates the specialized database schema that simulates user




(a) Workload a



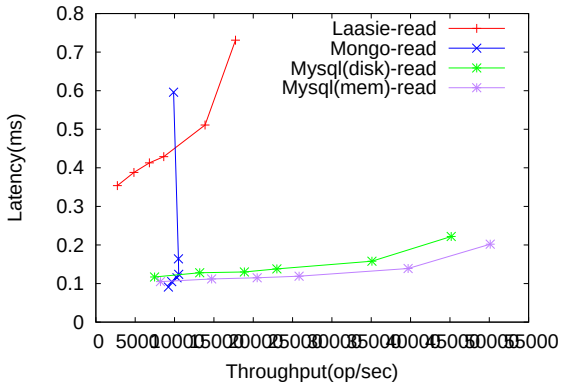

(b) Workload b



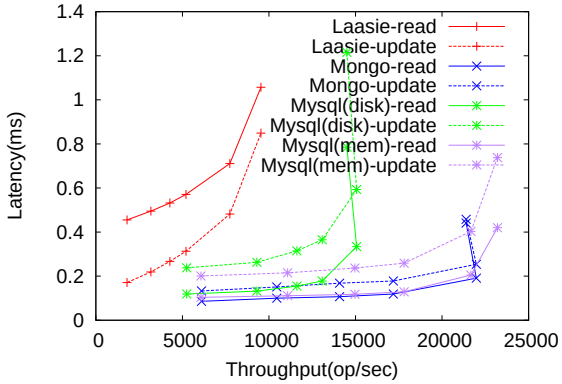



(c) Worklaod c

Figure 5.1: YCSB Results




(a) Workload d




(b) Workload f

Figure 5.2: YCSB Results

manipulations of a shared application state. These tests highlight two optimizations in Laasie:

1. Laasie clients maintain the local cache of the application state thus receiving only the updates which are required in addition to evaluate the reified application state at the client side.

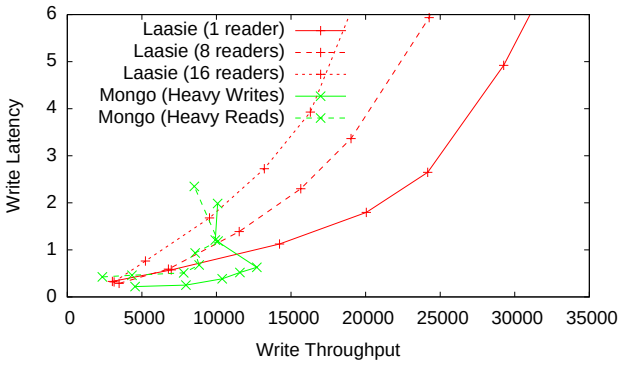
2. Read/Modify/Update operations are performed by composing the transformation to the collection of AST's maintained by Laasie server, rather than performing the update at client side.

Both systems (i.e. Laasie and MongoDB) were tested by instantiating multiple client threads to perform read and write operations. For Laasie, to simulate client-side caching, each client thread taken on the role of reader or writer. For MongoDB, client threads acted as both client reader. The ratio of read requests to write requests was varied from 10% to 90% (Heavy Reads, and Heavy Writes respectively).

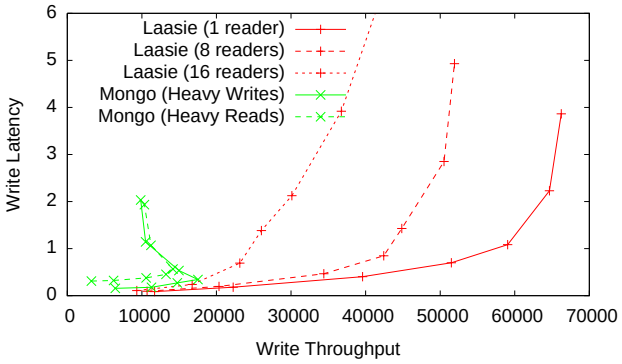
### 5.2.4 Results

Figure 5.3 demonstrates the relationship between write throughput and write latency. MongoDB achieved the throughput between 10 and 15 thousand operations per second before becoming saturated. Laasie achieved as much as 2-4 times the throughput on spreadsheet and paint benchmarks. Although due to limited support for string manipulation, performance suffered on the text editor benchmark.

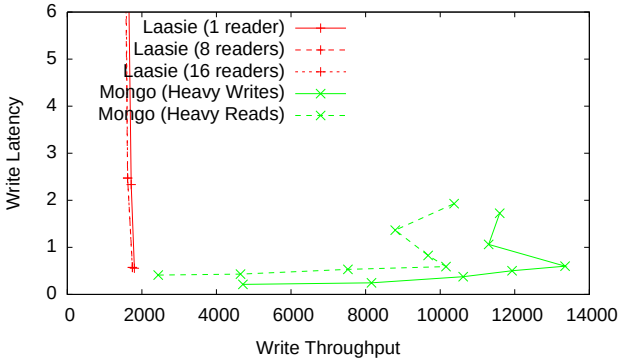
Figures 5.4 show the benefits of the (trivially supported) client-side caching and coherency capabilities of Laasie. These figures show the per-client latency to obtain fresh state. As refreshing the state requires a full copy in MongoDB, we obtained an equivalent per-client latency based on maximum observed read throughput of the server. The key takeaway from these figures is that the server side cost of updating a client to the latest state is not a blocking factor - even at 32 clients, the performance of the server is unchanged.




(a) Excel

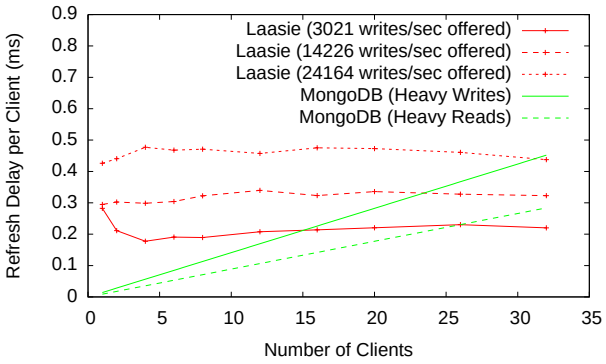



(b) Paint

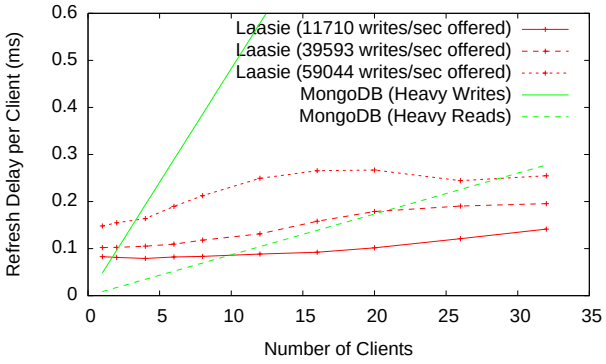



(c) Text

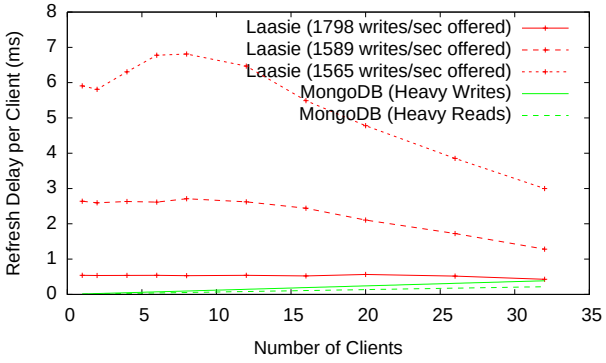
Figure 5.3: Latency v/s throughput benchmark comparison of Laasie and MongoDB




(a) Excel




(b) Paint




(c) Text

Figure 5.4: Latency v/s throughput benchmark comparison of Laasie nad MongoDB

# Chapter 6

# Conclusion

Specialized database solutions address some of the key issues which are not addressed efficiently by traditional relational database solutions. Storage and efficient retrieval of unstructured data stands out among all these issues. By exposing a limited or specialized API, each specialized solution makes strong assumptions about the application workloads. A tighter coupling with the application's workload allows data management system to provide improved performance, scalability, and ease of management. But this also reduces the system's capacity for post-hoc optimizations. Moreover, maintenance of several data management systems at application back-end can be costly, labor-intensive task.

The thesis argued for building block databases by proposing Abstract Syntax Tree (AST) as a database building block. AST databases (ADBs) stores the sequence of updates performed over the database state in its Data Flow Graph (DFG). The most recent database state can be obtained by evaluating the sequence of ASTs. ASTs can be coupled with any domain specific data manipulation language, decoupling an application's semantics from the underlying data management engine. Much like the data manipulation language, ASTs are amenable to program analysis techniques (such as static program slicing) which permits a great deal of flexibility in how the data is stored, accessed, and manipulated. One of the motivating use cases for AST databases is the collaborative web applications. Application's evolving requirements, and unpredictable growth often forces developers to restore to hybrid infrastructures cobbled together from distinct data management systems.

The thesis also presented the prototype of a document store AST database, called Laasie, which demonstrated the feasibility of AST databases in general. Laasie is specialized for ADB's primary operations like appending new updates,

accessing data by static program slicing, and AST rewrites. We evaluated Laasie against commercial and open source data management systems like MySQL, and MongoDB. The evaluation showed that Laasie's performance was within an order of magnitude of commercial systems on the workloads which are not optimized for use with Laasie. Also, on collaborative application benchmark (the main use case of ADBs), Laasie outperformed and/or showed better scaling characteristics than a commercial document store.

# Bibliography


[1] D. J. Abadi. Consistency tradeoffs in modern distributed database system design. *Computer-IEEE Computer Magazine*, 45(2):37, 2012.

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The lorel query language for semistructured data. *International journal on digital libraries*, 1(1):68–88, 1997.

[3] S. Abiteboul and V. Vianu. Collaborative data-driven workflows: think global, act local. In *Proceedings of the 32nd symposium on Principles of database systems*, pages 91–102. ACM, 2013.

[4] S. Agarwal, D. Bellinger, O. Kennedy, A. Upadhyay, and L. Ziarek. Monadic logs for collaborative web applications. WebDB, 2013.

[5] D. Axmark and M. Widenius. Mysql introduction. *Linux Journal*, 1999(67es):5, 1999.

[6] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 325–340. ACM, 2013.

[7] D. S. Batory, T. Leung, and T. Wise. Implementation concepts for an extensible data model and data language. *ACM Transactions on Database Systems (TODS)*, 13(3):231–262, 1988.

[8] P. A. Bernstein, C. W. Reid, and S. Das. Hyder-a transactional record manager for shared flash. In *CIDR*, pages 9–20, 2011.

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

[10] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 1995.

[11] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 35–46. ACM, 2011.

[12] S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *VLDB*, pages 1–10, 2000.

[13] C. Chodorow. Introduction to mongodb. In *Free and Open Source Software Developers European Meeting (FOSDEM)*, 2010.

[14] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 1. ACM, 2012.

[15] M. Corp. Office 365. http://office.microsoft.com/.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[17] J. Dittrich and A. Jindal. Towards a one size fits all database architecture. In *CIDR*, pages 195–198, 2011.

[18] Dropbox, Inc. Dropbox. https://www.dropbox.com.

[19] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *ACM SIGMOD Record*, 18(2):399–407, 1989.

[20] S. Even. *Graph algorithms*. Cambridge University Press, 2011.

[21] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[22] Google. Google docs. http://docs.google.com.

[23] T. J. Green, M. Aref, and G. Karvounarakis. Logicblox, platform and language: a tutorial. In *Datalog in Academia and Industry*, pages 1–8. Springer, 2012.

[24] T. Grust and A. Ulrich. First-class functions for first-order database engines. *arXiv preprint arXiv:1308.0158*, 2013.

[25] J. Han, E. Haihong, G. Le, and J. Du. Survey on nosql database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pages 363–366. IEEE, 2011.

[26] M. Heinrich, F. Lehmann, F. J. Grüneberger, M. Gaedke, T. Springer, and A. Schill. Enriching single-user web applications non-invasively with shared editing support. *Science of Computer Programming*, 2013.

[27] J. Hidders and J. Paredaens. Goal, a graph-based object and association language. 1993.

[28] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pages 11–11, 2010.

[29] O. Kennedy, Y. Ahmad, and C. Koch. Dbtoaster: Agile views for a dynamic data management system. In *CIDR*, pages 284–295, 2011.

[30] O. Kennedy and L. Ziarek. Barql: Collaborating through change. *arXiv preprint arXiv:1303.4471*, 2013.

[31] K. Lellahi and V. Tannen. A calculus for collections and aggregates. In *Category Theory and Computer Science*, pages 261–280. Springer, 1997.

[32] M. Levene and A. Poulovassilis. The hypernode model and its associated query language. In *Information Technology, 1990.'Next Decade in Information Technology', Proceedings of the 5th Jerusalem Conference on (Cat. No. 90TH0326-9)*, pages 520–530. IEEE, 1990.

[33] R. G. Miller Jr. Priority queues. *The Annals of Mathematical Statistics*, pages 86–103, 1960.

[34] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.

[35] K. Ostrowski and K. Birman. Storing and accessing live mashup content in the cloud. *SIGOPS Review*, 44(2), Apr. 2010.

[36] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 165–178. ACM, 2009.

[37] F. Pizlo, L. Ziarek, E. Blanton, P. Maj, and J. Vitek. High-level programming of embedded hard real-time devices. In *Proceedings of the 5th European conference on Computer systems*, pages 69–82. ACM, 2010.

[38] E. Redmond and J. Wilson. *Seven Databases in Seven Weeks*. Pragmatic Bookshelf; O'Reilly, 2012.

[39] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[40] M. Shapiro and N. Preguiça. Designing a commutative replicated data type. *arXiv preprint arXiv:0710.1784*, 2007.

[41] P. Shyamshankar, Z. Palmer, and Y. Ahmad. K3: Language design for building multi-platform, domain-specific runtimes. In *International Workshop on Cross-model Language Design and Implementation (XLDI)*. Citeseer, 2012.

[42] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era:(it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.

[43] C. Strauch, U.-L. S. Sites, and W. Kriha. Nosql databases. *URL: http://www.christof-strauch. de/nosqldbs. pdf ( 07.11. 2012)*, 2011.

[44] H. Tam Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi. Logbase: A scalable log-structured database system in the cloud. 2012.

[45] The Apache Software Foundation. Apache wave. http://incubator.apache.org/wave/.

[46] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[47] L. Wong. Kleisli, a functional query system. *Journal of Functional Programming*, 10(1):19–56, 2000.

[48] L. Ziarek, K. Sivaramakrishnan, and S. Jagannathan. Partial memoization of concurrency and communication. *ACM Sigplan Notices*, 44(9):161–172, 2009.