Netherlands Forensic Institute
Ministry of Security and Justice

Scalable performance for a forensic database application

Mattijs Ugen

April 18, 2013

Abstract

As digital forensic investigations deal with more and more data, the NFI foresees scalability issues with the current solution in the near future. Following the global trend towards distributed solutions for 'Big data' problems, the NFI wants to find a suitable architecture to replace the currently used XIRAF system. Using experimental implementations on top of a selection of distributed data stores, we present query performance timings in three different scaling dimensions: cluster size, working set size and the amount of parallel clients. We present that scaling characteristics for parallel clients show a linear trend, but proves hard to measure for the other dimensions. A distributed search engine architecture proves the best candidate for the NFI, warranting closer investigation in that area for a real-world deployment.

Acknowledgements

This work would not have been possible without the suggestions, feedback, cooperation and patience of a great number of people. First and foremost, many thanks to my supervisors Djoerd Hiemstra and Robin Aly at the University of Twente and Harm van Beek at the Netherlands Forensics Institute; their carefully directed questions and valuable feedback have shaped this work considerably. Thanks as well to my other coworkers at the Netherlands Forensics Institute for creating a relaxed, open and stimulating work environment. Last but not least, I would like to thank my friends and family for their everlasting support and encouragement.

Contents

Glossary

Image A file containg a bit-exact copy of the digital content of a device.

NFI *Netherlands Forensics Institute*, institute of the Dutch department of security and justice dedicated to forensic investigations and research.

Trace A singular piece of digital evidence, representing a concept like a file, phone call or chat message. See also Section 1.1.2.

Working set The subset of data present in a data store that is the subject of current use. Data *not* in the current working set could be for archival purposes or become part of the working set in the future.

XIRAF *an XML Information Retrieval Approach to digital Forensics*, a system designed to extract digital forensic traces from device images in an automated manner and allow users to filter these traces.

1 Introduction

The art of digital forensics consists of extracting and interpreting traces from digital material. Using specifications and reverse engineering of both physical devices and software components, the NFI is capable of turning a digital data set into a large collection of digital traces representing a wide range of concepts like (deleted) files, pictures, email messages, telephone call logs and processes running at the time a device was seized. The result of the extraction process is stored in a software system known as XIRAF [5] that enables users to search the total set of digital traces for items or activities that allow them to answer specific questions stated by clients.

The use of digital equipment is continuously increasing as the size of memory and storage in phones, personal computers and other devices still scales with Moore's Law [24]. As such, the amount of data processed by XIRAF is ever increasing, with an increased number of traces extracted from this data as a result. Given an increasing amount of traces stored in the database, XIRAF is forced to scale up to be able to cope with the pattern of use. The design of the current system does not allow for a future-proof way to scale up, requiring a bigger machine to process more data. To be able to handle the increasing amount of digital material processed and be more future-proof, the NFI is redesigning the system used for digital data analysis, of which the database layer is an important component. If the system as a whole would be able to scale out—be able to handle more data by adding more machines to a clustered setup—an ever increasing amount of data could be handled by increasing the total size of a cluster of machines rather than needing to replace machines with more powerful ones. Combining this with the fact that the increase in cost of server hardware is generally exponentially related to an increase in processing power, being able to scale *out* a system design is a desirable property.

The advent of the term "Big data" has not only revolutionized the way in which large data sets are analyzed, but has also sparked the development of distributed database systems to store large data sets in a structured manner. Issues with the traditional relational model surfacing when applied in a distributed environment are often dealt with by dropping the relational model in favor of a one that is often simpler and more performant in a distributed environment. Not supporting the relational model allows the use of alternative data and query models, making the choice of technology complicated as non-relational distributed data stores are not as easily compared as relational database products. Choosing the best database layer architecture now requires research and experimentation.

With this work, we contribute an evaluation of database technologies for a particular type of context to a field that makes comparison of available options tough due to differences in focus and features.

The rest of this report is structured as follows. The context of the application from which this work is motivated is described in Section 1.1. Previous work related to the context and the direction the NFI is heading is provided in Section 1.3. The research questions that have driven this work are provided in Section 1.2. The systems that have been selected for evaluation following the particulars of previous work are described in Section 2. Section 3 describes the way in which is the data and queries are translated onto the systems under evaluation. Methods and the experiment setup are provided

in Section 4. Finally, Sections 5, 6 and 7 provide the results, conclusions and a discussion of the work.

1.1 Context under investigation

Although this work aims to be a survey of general database technologies and scaling characteristics, it is motivated from the context of a digital trace database deployed at the NFI. The research context is currently known as XIRAF [5], an information system storing large amounts of digital forensic traces used by investigators to gather digital evidence. The XIRAF system is currently being evaluated and redesigned to be more future-proof and scalable. Expected data volumes and throughput characteristics play an important role in the design of the new system, which reflects in the statements made in this report.

The results and conclusions of this work are not aimed directly at XIRAF's successor, but XIRAF's particulars are taken into account when evaluating characteristics of database technologies. To put statements about the system under development into perspective, a number of high level requirements for the database subsystem of XIRAF's successor are listed below, followed by a more detailed description of the data and query models.

1.1.1 High level requirements

The main motivation of this work is the increase in size the material that is processed in digital forensic investigations. As the size pertains to both the number of cases processed per year and the amount of data that is associated with a case, the NFI expects the scale of things to become a problem for the current solution sooner rather than later. The high level requirements for XIRAF's successor are described in detail in unpublished work at the NFI. For the scope of our work the following requirements are taken into account:

Scalability: The system should be able to scale 'horizontally', allowing both the database throughput and size to increase at least linearly by adding more nodes to the system. Additionally, the system is to be used by many concurrent users.

No single point of failure: Any node that experiences a hardware or software failure should not cause the system as a whole to become unavailable or corrupted.

Data model: The system should be able to handle the data model that is currently used with XIRAF, the details of which are provided in Section 1.1.2.

Query capabilities: Apart from retrieving objects by their unique identifiers, the system should be able to search for sets of traces that match a set of predicates, the details of which are provided in Section 1.1.3.

1.1.2 Data model

The data of interest for this work is the result of the analysis of a piece of digital evidence, usually an bit-for-bit image of a physical device. The analysis process creates an annotation object for every trace extracted from the raw evidence data. An annotation is composed of a set of properties describing the trace it refers to and its relation to other traces. The complete set of annotations is

called the *meta data* of a piece of digital evidence. A directory on a file system, for example, is annotated with properties like the name of the directory, the last time it was accessed and its parent directory on the file system. A file on that same file system is annotated with properties much like that of a directory, but also with a descriptor that states where on the image the file resides. This descriptor allows XIRAF to retrieve the original file the trace was created from, also called the *content* of a trace. Every trace is assigned an identifier that is unique within the meta data for any image.

References to another traces turn the complete set of annotations into a tree data structure. The meta data that is handled by XIRAF is expressed in the eXtensible Markup Language (XML) [8]. Although references could potentially turn the data set into a graph—making a symbolic link on a file system refer to the trace it links to, for example—this is not permitted by XIRAF's data model specification. Preserving the presentation and interchange format is not necessary, the tree structure encoded as particular child elements in XML, *is* of importance. As such, a single trace can have an various number of distinct properties, each with its own typed value. Possible value types are *string*, *decimal number*, *integral number*, *time stamp*, and *boolean*. XIRAF's data model defines a fixed set of trace types, such as "file", "phone call", "deleted" or "registry entry". Furthermore, a trace can have multiple types, which would be the case for a trace representing a deleted file—residing in a recycle bin at the time of analysis—having both "file" and "deleted" types. The properties assigned to a trace are predefined for each possible trace type and can be optional. Appendix A.1 provides an example of a data set encoded as an XML document.

1.1.3 Query model

In contrast to a service like Twitter, where a user will often query the system for something specific, the question that will yield an investigator with the result that 'breaks the case' is often not know beforehand. XIRAF is often used to sieve through data, continuously narrowing the view of a data set to the point where a user will investigate the digital traces left in view. To accommodate this use case, an investigator is able to ask arbitrary questions regarding the digital trace data. The system has a very rich query model, allowing a range of different predicates to be expressed by the user:

Retrieval by unique identifier: Retrieves digital traces by a known unique identifier.

Match by property value range: Retrieves digital traces that have a property whose value matches a given range. To find all files modified in a particular week, a query could be issued for files whose property "modification date" is between the start and end dates of that week.

Retrieval by structure: Retrieves digital traces from the structure mentioned in section 1.1.2. An example would be to retrieve the children from a trace representing a directory on a file system, effectively retrieving a list of the files and the subdirectories in the directory trace.

Match by text property: Retrieves digital traces that have a textual property exactly matches, starts with, ends with or contains a given string of characters. Finding emails sent to anyone at the domain "example.com" for example, would be expressed as a text property query for traces of which a property named "recipient" ends with "example.com".

Match by full text query: Retrieves digital traces that match a full text query, such as finding text documents containing the phrase "this is an example phrase".

Match by trace type: Retrieves traces of a particular type, such as "file" or "email attachment".

The query types listed above do not cover the entire use of the current system, but form a representative set of features from the point of view of the database component. It should be noted that

these query types are used as filters and the system allows for boolean combinations of multiple filters. The system is also able to produce the byte sequence a trace is based on—the actual picture of a file trace representing a picture file, for example—but this is out of the scope for the database. This feature is assumed to be the responsibility of a different subsystem.

1.1.4 XIRAF query language

The queries issued to XIRAF are expressed in a domain-specific query language. In most cases, queries consist of a collection of filters that reduce the total set of traces in a data set to a subset that is of interest to the user. Examples are filters that restrict the result set to all files that end with *".pdf"* or email sent from account *"john@example.com"*. The collection of filters is provided as a hierarchical structure, where each filter is applied to the result of the filter it contains. The inner most filter is defined to be applied to the full data set. Note that filter expressions are commutative: reordering the filters in a hierarchical query has no influence on the result of the query.

Aside from filters, the language allows expressions pertaining to the structure of the data set. As the structure of the data set is a tree of traces, structure queries are issued as expressions on one of four axes: *ancestor*, *parent*, *child* and *descendant*. Their semantics are equivalent to their dictionary meaning: child elements of a trace that is a folder on a file system are the files and folders contained in that particular trace. The four axes are analogous to their semantics in the XPath language [11]. The semantics of a tree structure expression in the XIRAF query language differ from that of a filter expression. Instead of reducing the size of the result set by requiring that the properties of traces conform to particular predicates, a tree structure expression creates a new result set, possibly larger than the result set that would be created by the filters it contains. The new result set is created by finding the ancestors, parents, children or descendants of the traces in the result set of the inner filters. Note that tree structure expressions, unlike filter expressions, are *not* commutative: the result of a query changes if the location of a tree structure expression in the hierarchical query structure is changed.

Listing 1.1 Example query in the XIRAF query language

```
<has-text-property>
  <properties><property>
    <type>file</type>
    <property-name>name</property-name>
  </property></properties>
  <value>.pdf</value>
  <match>ends-with</match>
  <case-sensitive>false</case-sensitive>
  <has-long-property>
    <properties><property><property-name>content size</property-name></property></properties>
    <min>100</min>
    <has-type>
      <type>deleted</type>
      <unrestricted-document />
    </has-type>
  </has-long-property>
</has-text-property>
```

Listing 1.1 shows an example query in the XIRAF query language. The query contains three filters: the property *"name"* for the type *"file"* should end with *".pdf"*, the property *"content size"* should be greater than 100 and traces should have type *"deleted"*. In other words: the query defines a search for deleted PDF files larger than 100 bytes. The innermost filter in the example is applied to the full set of traces in the data set, expressed as `<unrestricted-document />` in Listing 1.1. Appendix A.2

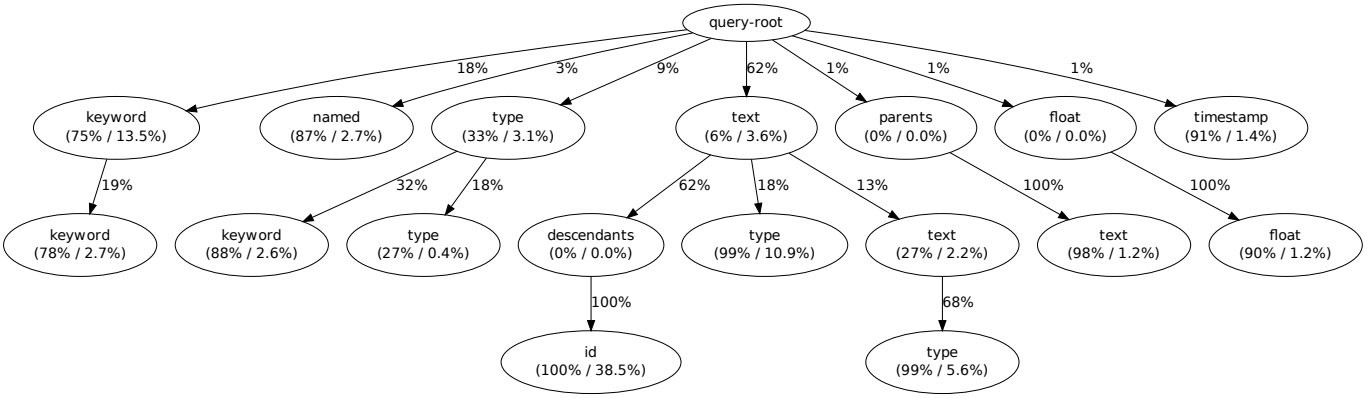

Figure 1.1 Query filter combination statistics (vertices and edges representing less than 1% of total query set have been omitted for readability).

provides additional examples using more complex features of the XIRAF query language.

1.1.5 Context analysis

Using application logs from XIRAF, usage statistics of the features in the query model are generated. As XIRAF queries are composed of a number of filters, the combination of different types of filters can be used to build a statistical model of the usage pattern of the database layer of the application. Figure 1.1 displays the result of a statistical analysis of around 250.000 queries issued by XIRAF in a two year period. The paths from the root of the tree to any other node represents a possible combination of filters found in the application logs. The path from the *query-root* to *timestamp*, for example, represents queries with exactly one filter, matching traces that have a property of type *timestamp* within a certain value. The percentages in the nodes represent the relative amount of queries that 'end' in that node, containing the filters of the types along the path to that node and the contribution of those queries to the total set that was analyzed. The percentages do not add up to 100% as a part of the data that make up the figure as a whole has been omitted for readability.

1.2 Research questions

Considering a future application that satisfies requirements as stated in Section 1.1.1, the NFI is interested to know what distributed data store architecture would suit the context best. As such, the main research question we aim to answer is the following:

1. Which distributed data store architecture is the most suitable for the context of the NFI?

As there are multiple facets to the suitability of a data store implementation to the context, some additional questions need to be answered in order to provide an answer to the main research question. The following questions flow from the high level requirements stated in Section 1.1.1:

2. What features are required of a distributed data store to be applicable to the provided context?
3. How does query performance of a distributed data store relate to the size of a cluster?
4. How does query performance of a distributed data store relate to the working set size?
5. How does query performance of a distributed data store relate to the number of clients using the system?

Following findings in previous work, a number of data stores are selected to evaluate according to the research questions stated here. Section 2 provides a brief overview of these systems. Sections 3 and 4 provide implementation details to apply the selected data stores to the provided context and the experiment setup to evaluate their performance respectively.

1.3 Previous work

Creating and dealing with large databases has been on the agenda of the industry and research community for about as long as databases have existed. With the advent of the 'Big data' hype, however, things have taken a dramatic turn, with multiple Internet scale companies to publish radical solutions to scale issues. The following provides an overview of work surrounding the distribution of data for a number of applicable data models as well as the description of a number of available highly scalable architectures.

1.3.1 Data models

The relational model is seen as the most used data model in the world today. XIRAF currently uses a relational database, even though the data model of the application is not strictly relational, as mentioned in Section 1.1.2. Scaling and distributing a relational data set has mostly focussed on optimizing partitioning of a data set and optimizing joining partitions as part of a JOIN operation. The data normalizations commonly made in the relational model often cause an increased amount of JOIN operations, which causes network traffic when the two operands of the operation exist at different hosts. Table partitions can be made both 'horizontally' and 'vertically', each with their own particulars. Strong consistency guarantees are cumbersome to keep in a distributed environment, while the number of vertical partitions is limited by the number of columns.

Graph databases have seen a increase in interest in the past year with the growth of popular social networking applications like Facebook and LinkedIn. Due to the scale of these applications, new technologies have been developed to deal with the amount of data. Google published a data processing model dedicated to graphs by the name of "Pregel" [22] after concluding that the Map/Reduce paradigm does not fit well with many graph data processing algorithms. Although graph database technology has demonstrated applicability to large scale in the past years, the query interface of graph databases is not geared towards our application area [59].

Considering that the data model of the application context for our work is a tree, there is much to say about modeling and partitioning tree data. As trees are almost synonymous with XML in the academic world, many interesting works about modeling tree data is carried out in the context of XML databases. In most situations, the XML structure is flattened into a collection of nodes, where a node label determines the place of the node in the tree, allowing the original tree structure to be reconstructed from the flattened data. Many labeling schemes have been developed with various foci, ranging from the standard *(pre, post)* tuples [16] to update-efficient ORDPATH [26] and CDQS [20]. Although XML databases are often implemented on top of row-oriented data stores, MonetDB/XQuery [7] proves that efficient XML processing is not limited to row stores.

1.3.2 Distributed architectures

Distribution of data is far from simple, as many works have been published describing different ways in which with corresponding pros and cons to partition a data set for use in a distributed environment. The following sections describe data store architectures that have been shown to highly scalable and as such are interesting for our context.

Traditional database technology
Following the success of the NoSQL landscape, ORACLE enhanced the popular relational database MySQL with the possibility of running in a horizontally scaling clustered setup [52]. The implementation shows horizontal scalability in the number of sustainable SQL UPDATE operations per second and low latencies in ORACLE's own tests. The issues concerning the relational data model and the normalization generally associated with it mentioned above will however likely still cause a performance hit for the setup.

Distributed trees structures
Although using the tree data model for database solutions dates back to to early 1970s [12], the distribution of data modeled as a tree is something sparked by the widespread adoption of XML. To facilitate processing of data provided in XML format, various XML databases have been developed over the years, although many are mostly used as a research object than a product [17, 23]. Not much work has gone into designing an XML database that is distributed. Pagnamenta published a design of a distributed XML database [27], using a heterogeneous collection of relational databases and an abstraction layer to form a single distributed system. Partitioning a tree data structure other than a horizontal partitioning of its flattened representation is considered to be a hard problem, as demonstrated by works like that of Bremer et al. [9] and Kido et al. [18].

Querying an XML database is most commonly done using the standardized XQuery language [6], providing powerful procedural query execution to users. For simpler use cases, XPath [11], a subset of XQuery, can be used, allowing users to express the requested set of elements of an XML document as a path along various axes.

Column family store
Initially published by Google in the form of Bigtable [10], the so-called *column family store* is a data store focussed around denormalizing data into rows with a variable amount of columns. Being column-oriented, a number of optimizations unique to column-oriented databases are possible the column family store architecture. Originally developed for relational data stores that utilize a column-oriented storage technique, methods like *run-length encoding* [1] and *block iteration* [37, 38] are applicable to the Bigtable architecture and allow for performance optimizations. Not dealing with data entities as the relational model does, *late materialization* [2] is the default mode of operation for the Bigtable architecture as the client side explicitly states the columns that need to be retrieved, boosting performance for various use cases.

The data model of a column family store is deliberately quite simple. At the top level, the data model defines a table with a number of column families. Each row in the table is identified by a key that is a sequence of characters and contains variable number of columns in any of the defined column families for the table. Distribution of data is done by splitting the range of the keys in the table into consecutive regions and assigning regions to data nodes in the distributed system. Regions can be split and merged should mutations on the table increase or decrease their size. The storage model utilized by a column family store is reminiscent of the log-structured merge tree [25], dubbed an *SSTable* or sorted string table by the authors of Bigtable.

Querying a column family store can be done using one of two operations: GET or SCAN. A GET operation takes the key of the row and a specification of the columns to be retrieved. A SCAN operation can take a more complicated form and allows a client to retrieve values from the table as a stream.

As Bigtable has not been made publicly available by Google, multiple implementations of the design have been created in the form of HBase [57] and Hypertable [48]. Although both implementations are designed to run on their own infrastructure—HDFS for HBase [58] and a DFS Broker capable of running on a range of different file system implementations for Hypertable [48]—both stay very close to the Bigtable architecture.

Distributed hash table
Distributed hash tables have been around for quite a while as a research topic systems like CAN, Chord, Pastry and Tapestry [30, 33, 31, 36], serving mostly to be a fault-tolerant routing and discovery mechanism for content on the Internet. More recently, distributed hash tables have seen increased interest with the publication of Dynamo [13], a highly available scalable data store designed by Amazon, turning the concept into a fault-tolerant high performance data store. Although Dynamo is described as a purpose-built system for the requirements of Amazon, a number of other big Internet companies have created Dynamo-like systems to satisfy their needs, like Facebook's Cassandra [19] and LinkedIn's Project Voldemort [14]. One of the nice features of a distributed hash table is that the design does not need a 'master' role; all nodes in a cluster are aware of their place on the key ring and can identify the position of any requested value by hashing the request's key. The configuration of the cluster and communication of data such as the information of the nodes on the ring is done using a gossip-like protocol. Following the publication of Dynamo, researchers at Amazon have continued research efforts on distributed hash tables by designing an efficient gossiping algorithm known as "Scuttlebutt" [34].

From an architectural point of view, a distributed hash table is designed to be able to answer a single type of question: retrieving the value associated with some key. Some implementations of the architecture offer extensions on the basic functionality by allowing to retrieve a particular part of the

value of saving additional information with the values that allow extra operations like finding related keys or values.

Research on enabling searching in a distributed hash table is not new. Generally, approaches towards this can be categorized as one of two techniques: adding overlay structures to the distributed hash table or preserving order between keys rather than using uniform hashing. Overlay structures such as *skip graphs* [4], *prefix trees* [29], *range search trees* [15] and *space-filling curves* [32] have been shown to be successful in augmenting distributed hash tables with range search capabilities. Order preserving key assignment trivially enables the ability to retrieve keys in their natural order, though defeats the point of a distributed hash table and introduces a number of problems as described by DataStax for Cassandra [44].

1.3.3 Available systems

The many advancements in database technology made over the years do not all make it into available systems overnight. A small list of available distributed database systems is provided here with a indication of their architecture and notable features.

Cassandra: Inspired by Dynamo, Cassandra is a distributed hash table with the addition of supporting structure in the values associated with keys. Structured values allow operations on parts of the values stored in the system rather than retrieving it whole with a GET operation [19].

ElasticSearch: Distributed search engine using Lucene [54], achieves high availability and distribution by separating collections of documents into separate indexes and replicating parts on multiple servers [47].

HBase: Inspired by Google Bigtable [10] and part of the Hadoop ecosystem of Big data systems, HBase is a column family store deployed on Hadoop's HDFS [58].

Hypertable: Like HBase, Hypertable is an implementation of the Bigtable design, capable of running off of a range of possible distributed or local file systems implemented in C++ rather than Java [48].

MongoDB: Document store distributing data on a user-supplied property with advanced query capabilities [39].

MySQL Cluster: A distributed version of the popular relational database engine, using horizontal partitioning and replication.

Neo4j: Distributed graph database, uses *property graphs*, indexes on property values, focused on graph analysis [51].

Riak: A distributed hash table augmented with indexes and links to annotate a stored value with meta data and relational information [41]. Enables searching through stored data through Riak Search [42].

Solr: Distributed search engine based on Lucene [54]. Developed in tandem with Lucene itself and in many ways comparable to ElasticSearch [55].

Voldemort: A pure key-value store used for value retrieval by key, with configurable back end storage layers [14].

As XIRAF has moved from an XML database in the form of MonetDB/XQuery [3] to a relational database at the time of writing [5], another solution for the next generation is to be devised. Based on the problems uncovered in this Section 1.3 and experience with both XML data stores and relational data stores, the NFI is currently aiming for a highly scalable solution. Section 2 elaborates on the data stores that have been selected for our work.

1.3.4 Related work

Determining the best data store architecture to use for an application context is a problem often seen by industries around the world. A recent study by Rabl et al. [28] is interesting for our work, as they make a comparison of contemporary 'NoSQL' key-value stores for the context of Application Performance Management systems. Although very insightful in the fact that horizontal scalability is visible in their experiments, the study mostly measures maximum sustainable throughputs for simple operations rather than complicated requests as issued by XIRAF. As complicated requests need to be constructed from the simple ones offered by an architecture as a key-value store, the study provides valuable input in the choosing of viable candidates for the provided context.

2 Systems under investigation

If a usable overview is to be created, the systems that are to be investigated should be applicable to the context outlined earlier and differ on a fundamental level. In order to expose strengths and weaknesses of different architectures, four data store implementations have been chosen that differ in the way data is distributed among nodes and/or in the way queries are evaluated. These choices have been made following the particulars described in Section 1.3 and preliminary experimentation with a number of data store products that are freely available.

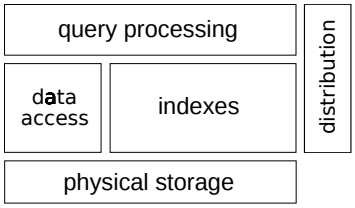

Figure 2.1 Components required of a database system.

Four implementations have been chosen to conduct experiments with: *Cassandra*, *ElasticSearch*, *HBase* and *MongoDB*. The implementation details that play a role in their performance characteristics for the scope of our work are explained below for both the distribution of data among nodes and the manner in which queries are executed. Figure 2.1 depicts five basic components needed by a database system to be applicable to the provided context:

Physical storage: a means of persisting data stored in the system;
Data access: a means of storing collections of data in a way that allows a user of the system to retrieve elements of it by their identifier—or primary key;
Indexes: a means of finding a subset of the data set that match certain criteria;
Query processing: A means of producing a subset of the data given criteria provided by a user;
Distribution: A means of distributing data, indexes and query processing in a way transparent to the user.

2.1 Cassandra

Cassandra is a distributed hash table that goes beyond the concept of a pure key-value store. Where most key-value stores available at the time of writing provide a simple GET and PUT interface based on the key of a key-value tuple, Cassandra allows to define a structure in both the key and the value that is stored in the form of columns. Given a key, the structure in the stored value can be used to instruct Cassandra to retrieve only a specific part of the value. The semi-structured data of values can be used to build indexes that allow the retrieval of a slice of this index without needing to transfer it whole. Although Cas-

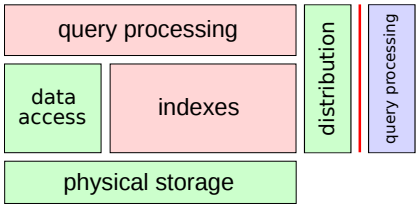

Figure 2.2 Basic component overview of Cassandra.

sandra natively supports secondary indexes on data, these are very limited in functionality and not recommended for use with columns that have a high number of distinct values [45]. More specifically, Cassandra's secondary indexes do not facilitate range matches for numeric values and containment matches for string values, both of which are required by the provided context.

Cassandra defines no special roles to nodes that participate in a cluster, as is common with distributed hash tables. The key space for Cassandra is the value range of the MD5 cryptographic hash algorithm, a widely used uniform hash function that has a 128 bit output. Each node is made responsible for a part of the available key space. Using a uniform hash algorithm like MD5 ensures uniform distribution of data over the available nodes, but also removes all relation between values that are consecutive in their natural order. Partitioning data in order would potentially allow the retention of subsequent values and is technically possible with Cassandra, though the feature has been deprecated as this approach has led to operational issues [44].

A collection of data in Cassandra is called a *column family*. A column family contains *rows*, each identified by a *key*. Rows contain a list of *columns*, each with a column name and a column value. Keys, column names and column values all allow a type specification to be defined for them, making sure the keys are treated as, for example, integral numbers or composites of multiple types. Multiple column families can optionally be grouped in *key spaces*.

Figure 2.2 shows the basic components mentioned above. Both query processing features offered by Cassandra and the indexes provided natively are considered inadequate for the query model as described in Section 1.1.3 and have consequently been colored light red. An additional query processing component is depicted on the client at the other side of a red bar, indicating a client/server side boundary. This additional query processing component is needed to fill in the deficiencies in the features provided by Cassandra itself.

Cassandra version 1.1.8 was used for our experiments.

2.2 ElasticSearch

Built on Apache Lucene [54], ElasticSearch [47] is a document store designed to be a distributed search engine. The distribution of data in ElasticSearch is done by splitting a Lucene index into a collection of subsets of the documents in an index called *shards*, which ElasticSearch uses as the unit of distribution. To facilitate growth of an index over time, common practise is to *overcommit* shards, creating more shards than needed, allowing the system to move smaller portions of data around when needed. High availability of the data in ElasticSearch is ensured using replication, copying shards to multiple nodes in a cluster to allow operations on an index when a machine becomes unavailable. A shard is saved to disk as-is; the files representing the shard on disk are regular Lucene index files.

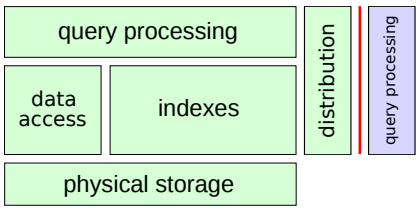

Figure 2.3 Basic component overview of ElasticSearch.

Queries are executed on all shards of an index, merging the results from all shards into a single re-

sult set before returning the complete result set to the client. An index in ElasticSearch contains both the documents provided by the user and data structures to facilitate queries on any property of those documents. Figure 2.3 shows the basic components mentioned above. As the indexing structures created by ElasticSearch facilitate a large number of different queries applicable to the provided context, all basic components have been colored green. For queries not natively supported by ElasticSearch, an additional query processing component is added on the client side at the other side of the client/server boundary.

ElasticSearch version 0.19.11 was used for the experiments in our work.

2.3 HBase

HBase is closely modeled to Google's Bigtable [10], aiming to be a highly available distributed data store supporting both key-value-like data access and data scanning facilities for complex analytics tasks. The system is part of the open source Apache Hadoop ecosystem. Data in HBase is formatted in a way similar to Cassandra, associating a collection of columns with a particular key. The top level data structure is a *table* containing *rows*. A row is identified by a *key* and contains a variable amount of columns. The columns in a row are grouped by *column families*, which are predefined at the table level. An important distinction between Cassandra and HBase is the fact that HBase stores rows in the lexicographical byte order of the keys. Distribution of data in HBase is governed by the concept of *regions* in the available range of keys, which are distributed among nodes in a clustered setup. The HBase *master* can split a region in two should a region become larger than a user-defined threshold. The actual files HBase uses to store the regions are put on the Hadoop distributed file system, more commonly known as HDFS [58].

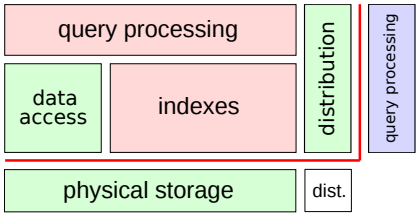

Figure 2.4 Basic component overview of HBase.

Figure 2.4 shows the basic components mentioned above. The query interface for HBase is limited to simple operations. Although the data scanning interface allows the user to supply a complicated filter to retrieve the data that is of interest, this feature is not suitable for ad hoc querying. This is covered more thoroughly in Section 3.4.2. Indexing structures are not natively supported by HBase, requiring the addition of data structures usable as such. Because of the deficiencies in features natively supported by HBase required by the provided context, both query processing and indexes have been colored light red in Figure 2.4. An additional query processing component is depicted on the client side. Also, as HBase is deployed on a distributed file system, the physical storage component can be located on a different host as well, as indicated by the additional red bar and distribution component, depicted by the *dist.* component at the physical storage level. When the distributed file system is deployed on the same machines as the region servers are, HBase communicates with the distributed file system software through a local channel rather than requesting data over the network.

HBase version 0.94.4 was used for the experiments in our work.

2.4 MongoDB

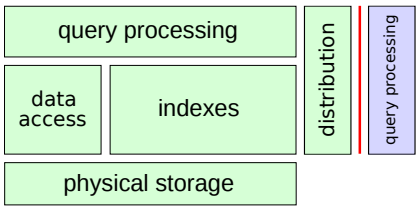

Figure 2.5 Basic component overview of MongoDB.

MongoDB is a schema-free document store, allowing any valid JSON-encoded data to be inserted in a document collection. The distribution of documents is done by using blocks of data called *chunks* and distributing these among nodes. In normal operation, MongoDB uses *replica sets*, a group of machines that contain the same data, to ensure high availability. The nodes involved in a replica set together form a *shard*, which can be used in a sharded setup, connecting many servers together to form a MongoDB cluster. A particular attribute (or composite of multiple attributes) called the *shard key* is supplied by the user to define the grouping of documents into chunks; documents with subsequent values for the shard key attribute will appear together in a chunk of data. Note that the meaning of a shard for MongoDB differs from that for ElasticSearch, where a shard is comparable with a chunk for MongoDB. The value of this required attribute for each document defines the shard the document belongs to. Should the size of a chunk exceed the maximum size—64 MB by default—the chunk is split in two, allowing the newly created chunks to be filled with more data. This behaviour differs from ElasticSearch, where the amount of shards in an index is fixed from the moment the index is created. A clustered setup is administered by a *config server*, which is a regular MongoDB database server issued the specific task of keeping track of the nodes and chunks of data in the cluster.

Querying a collection of documents is done by supplying a reference document containing predicates on attributes, defining the result as the subset of documents that satisfy the predicates in the reference document. If the query contains a predicate on the shard key, MongoDB will be able to make an assumption on the shards that need to be queried for the requested values. If not, a *"scatter gather"* technique is used, in which all shards of the cluster are involved. After gathering partial results from all shards, the partial results can be combined and returned to the client. The MongoDB *router server* is used as a query entry point for clustered setups, caching information from the *config server* that is responsible for administrating a MongoDB cluster and issuing requests for partial data to the nodes containing the data. MongoDB natively supports secondary indexes on attribute values, aiding in processing query requests.

Like ElasticSearch, the querying and indexing features of MongoDB are quite advanced. As such, all basic components mentioned above are colored green in Figure 2.5. For queries that cannot be processed with MongoDB's native features, an additional query processing component is added on the client side.

MongoDB version 2.2.2 was used for this work.

3 Solution designs

Considering the requirements provided in section 1.1.1, data needs to be translated to a format understood by the selected data stores. Both data formats and query capabilities vary greatly between the selected data stores, so a number of solutions were developed that allow the four data stores to be used in the provided context. The data structures for all data stores need to make sure queries can be answered when the query capabilities of the data store itself are not sufficient. Likewise, all selected data stores need to be able to process queries using the means provided by the manner in which data has been stored. Both approaches common to all implementations and techniques deployed for data store specifics are provided below. Implementations were created using features either unique to the data store in question or features the data store is optimized for and/or reportedly good at.

3.1 Generally applicable solutions

Although the four selected data stores differ on many levels, a number of solution details are reusable for all or are required by the context. The following two sections describe solution designs that are applicable to all four data stores.

3.1.1 Data structures

Given that searches for particular string values can be issued as *equality*, *prefix*, *suffix* and *containment* as well as either case sensitive or case insensitive, properties with string values are duplicated in various transformations. Supporting queries for exact matches is readily available in all data stores, provided that an index structure is present. Querying for values that start with a particular value is available too, as data stores allow to retrieve values that have a particular prefix. Querying for values that *end* with a particular value is less trivial, but is easily translated to a prefix query when both the stored and queried values are reversed. A similar solution is available for string values that are queried case insensitively. For this, both the stored and queried values and are converted to lowercase and again issued as exact or prefix queries. Containment queries for string values are more complicated. Considering that an index on a property with a string value does not support efficient retrieval of objects that *contain* a queried value—if this is possible at all—additional measures need to be taken. This feature is enabled by storing "edge n-grams"—a collection of substrings—of any string value alongside the original an approach inspired by the use of edge n-grams in the Lucene software library [56]. As a prefix match on any of the generated substrings is equivalent to the original string containing the queried value, the string containment requirement can be satisfied in this way. As long strings generate a large number of edge n-grams, the amount of space required to save all edge n-grams of all string type properties in a data set is high. To reduce the impact of this size explosion, the maximum size of edge n-grams has been set to fifteen characters. So, the first edge n-gram for a property value *"longer-than-average-file-name.txt"* is *"longer-than-ave"*, the second edge n-gram for

the value is *"onger-than-aver"*. The value will generate a total of eighteen edge n-grams, the last one of which is *"rage-file-name.txt"*. Considering that the n-grams are used for prefix matches, limiting the length of an edge n-gram requires that queried values are limited to the same length, as a sixteen character value can never be a prefix of a fifteen character edge n-gram.

Encoding the tree structure relations among traces is done by saving the sequence of ancestor identifiers for every trace along with the level in the tree on which the trace occurs. Ancestor identifiers are encoded as base 36 numeric values, delimited in the sequence by comma characters, usable as a string property for every trace in the data set. Using base 36-encoding for the numeric values of the trace identifiers provides a slight compression over base 10-encoded values and is conveniently available in the Java API for numeric type values. As an example, consider a file trace with identifier 23 that is a child element of a folder trace with identifier 10 whose parent is the root of the tree with identifier 1 such as in Listing A.1 in Appendix A. The *path* property—distinct from the file and folder path properties shown in the example—for for the file trace with identifier 23 is *"1,a,n,"*, encoding the location of the trace in the tree from the root to itself. As the file trace is on the third level of the tree, its *level* property is set to 3.

3.1.2 Query translation

All of the data stores that were selected in Section 2 are capable of retrieving a trace by a known identifier. For Cassandra, this is done by defining a column family with trace identifiers as the key and the set of properties and their corresponding values as the columns and values of a row. ElasticSearch supports the retrieval of documents by their *internal* document identifier, a number that is not under control of the user, but supplied by the underlying Lucene library and subject to change [53]. For the viewing of trace details from a result list, the internal document identifier is known as it is included in the result retrieved from ElasticSearch. For other cases, a search request for the trace with the *application* identifier equal to the requested value can be issued, resulting in a search result of size one. Retrieving a row containing the properties and values for a specific trace with HBase is equivalent to the approach used with Cassandra: by creating a table with the trace identifiers as keys and the properties and values as columns and values trivially satisfies the retrieval of singular traces. MongoDB, like ElasticSearch uses internal document identifiers, but unlike ElasticSearch, these identifiers are not subject to change. Once the internally defined document identifier is known, retrieval of the corresponding trace is possible with MongoDB's `findOne` command. Using the application identifier is equivalent to using MongoDB's internal identifier as long as the identifying field has an index associated with it.

Finding the descendants of a set of traces is achieved by looking up the path of the given set of traces and issuing a prefix query for every ancestor path. Finding children is possible in a similar manner, with the addition of requiring a particular level in the tree, exactly one greater than their parent traces. Considering the example folder trace with identifier 10 above, all of the files and folders contained by the folder can be obtained by issuing a prefix query on the *path* properties of all traces in the data set. Retrieving all descendants of the example folder trace is done by issuing a prefix query on the path property starting with *"1,a,"*. Similarly, all traces with that particular path property *and* a level property with a value of 3 are children of the folder trace. Note that the trailing comma is included because the value is used as a prefix for the path property. Should the trace with identifier 1 also have a child trace with numerical identifier 361—*"a1"* in base 36—the path of the child is encoded as *"1,a1,"*. The queried value *"1,a,"* is a prefix for the trace with path *"1,a,n,"*, but *not* for the trace with path *"1,a1,"*. Upon retrieving the ancestor paths of traces, the parent and ancestor identifiers

are readily available in the path properties of the traces—encoded as "*1,a,n,*", for example—requiring no further query requests.

The order among siblings in the tree structure, although present in the XML format that is specified by the XIRAF data model is lost with the tree path encoding scheme used here. Upon investigation of the corresponding query language specification, however, there is no means of reasoning about the order of traces in the tree structure. Investigators at the NFI have stated that the sibling order among traces is not significant for their work. As such, this loss of information is assumed to be permissible.

The XIRAF XML document in Appendix A is used as an example for the data structures in the sections below. The XIRAF query in Listing 1.1 is used as an example in how the data stores process a query.

3.2 Cassandra

Considering the components necessary for a data store to be usable for the provided context, Section 2.1 states that Cassandra lacks sufficient indexing and query processing capabilities. The following describes the data structures and query processing that were developed to satisfy the requirements described in Section 1.1.1 with Cassandra.

3.2.1 Data structures

To be able to retrieve complete traces with details, a single Cassandra column family is used with trace identifiers as row keys. Rows are formed by mapping canonical property names to their respective values, letting Cassandra take care of the byte sequence encoding of both names and values.

As Cassandra provides a relatively restrictive query interface, the trace data column family does not allow searching in the data. To facilitate searches in all kinds of data, a number of additional column families is created that index the trace data column family, one for each possible type of value. Each of the index column families is constructed by using the property name as the row key. Within each index row, column names of the compound form *(property value, trace identifier)* map to the trace identifier. The trace's identifier is included to ensure idempotent write operations on the index for traces with identical values for a property. Using the compound form saves the use of collection type values which require a *read before write*, potentially causing lost updates and not storing traces in the index in an environment where multiple processes write to the same index. Adding the trace identifier to the column name ensures a unique index entry for a trace's value, removing the possibility that another index update from a concurrently running insert operation will overwrite it. The index update operation is idempotent—repeating the action will not alter the index, only overwrite the identical value already present—so it can safely be repeated should an error occur without knowing whether the operation succeeded or not. By using Cassandra's own `Comparator` implementations for the corresponding value types, natural order—that is, -1 comes before 0 and 0 comes before 1—among various data types is guaranteed, allowing range queries on both text and numerical properties. Additional index structures are created for the purpose of particular query types. For trace type queries, a type index column family is created that uses a type's name as the row key, trace identifiers as column names and empty byte sequences as the column values. For tree structure expressions, a col-

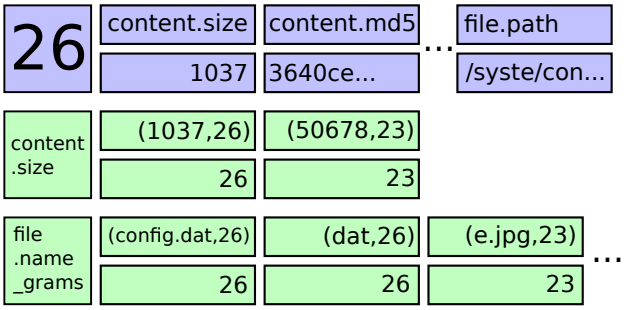

Figure 3.1 Data from the document in Listing A.1 as saved to Cassandra rows.

umn family consisting of a single row is created, using column names of the form *(path, level)*, where path is a sequence of ancestor trace ids with separators and level the depth of a trace within the tree structure. The values used for these columns are the identifiers of the trace that corresponds to the path in the column name.

Figure 3.1 shows a number of rows from the data that is saved to Cassandra. The top row shows the row that stores the trace with identifier 26 as a single entity, storing properties like *"content.size"* with their corresponding values. The trace identifier is used as the row key. The second row shows the index for the *"content.size"* property in the column family storing index entries for integral number types, containing the described compound column names with both the property value and the trace identifier, ordered by the property values. The third row shows a different index entry for the edge n-grams of the *"file.name"* property. Here too, the columns are sorted by the value that is store in it.

As a comparative measurement, a XIRAF analysis result XML file of 256MB, containing 370330 traces was loaded with Cassandra using the data structures above. After flushing the column families created for it to disk, the 370330-trace data set measured 2183MB in size, 8.5 times the size of the original data.

3.2.2 Query translation

As mentioned, Cassandra features a query interface that does not meet the requirements for the provided context. Using *column slices* and the combination result sets on the client side, the combination of multiple filters in a single query can still be made. The index column families created for each value type are used for their corresponding filters. A request for traces with a *"content.size"* property smaller than 100 bytes, for example, would be mapped to a column slice from the column family for integral number type properties, using the row with key *"content.size"*. As the columns are ordered by the value of the *"content.size"* property, a column slice from the start of the row up to the column that contains the last value before 100 retrieves all trace identifiers that satisfy the request. In the case of multiple filters, a subsequent column slice retrieving the trace identifiers for that filter can be intersected with the previous one, resulting in a logical conjunction of the two filters. For more filters, the process can be repeated.

Considering the example query in Listing 1.1, the pseudo code procedure in Listing 3.1 is used to satisfy the query with Cassandra:

Listing 3.1 Pseudo code procedure to satisfy example query in Listing 1.1.

```
Results1 = SliceStringIndex('file.name_reverse', 'fdp.', 'fdp.' + MAX)
Results2 = SliceIntegerIndex('content.size', 100, MAX_VALUE)
Results3 = SliceTypeIndex('deleted')

Results = Intersect(Results1, Results2, Results3)
```

"Slice" functions are defined as returning a set of trace identifiers contained in the part of the index that is requested. The requested slice is provided as the minimum and maximum values within the index. The string slice is issued for the reversed version the *"file.name"* property, using the reversed version of the requested value *".pdf"*. The maximum value for the string slice is set to the same value, extended with the 'highest' printable character, effectively specifying the last possible value starting with *"fdp."*. For the integral number slice, the minimum value is set to 100, as requested from the query. As the query states no maximum value, the highest possible number value is used as the maximum value, requesting all index entries between 100 and the end of the index. The requested type has no value range, so only the requested value *"deleted"* is needed for the last filter. Finally, the results of all index slices are combined into the requested result set by intersecting the intermediate result sets, leaving only the identifiers of traces that match all the requested filters. For the implementation of the query execution, the Hector [46] library was used, which takes care of marshalling objects to and from formats that can be transported over the wire to Cassandra.

3.3 ElasticSearch

As Section 2.2 mentions that ElasticSearch offers all of the basic components needed to be applied to the provided context, little features were created to implement the query execution engine on top of ElasticSearch.

3.3.1 Data structures

As the relations among traces can be encoded using ancestor paths, each trace can be saved as a separate document, translating traces in XML format to separate JSON-encoded documents. Because ElasticSearch is a search engine rather than a database, some settings need to be tweaked for it to leave some of the data intact. Using default settings, string type values would be analyzed for full text search, an approach that is not desirable for the properties in the meta data as string type properties will possibly get tokenized into separate words. By changing settings so all values are indexed without content filters, ElasticSearch is made more compliant to the context requirements. As the Lucene library on which ElasticSearch is built has support for value types other than text, no further actions are needed to save the data to ElasticSearch. After the XML to JSON transformation, a document as lister in Listing 3.2 is saved to ElasticSearch. Note that a number of properties added to facilitate certain query features as described in Section 3.1 have been omitted for brevity.

Listing 3.2 Trace encoded as a JSON document for use with ElasticSearch.

```
{
  "content": {
    "size": 1037,
    "md5": "3640ce36ca049c9cd81cf354cdfe5c76",
    "md5_grams": ["3640..." ... "...5c76"],
    "sha1": "4c5f05440b87dcf86e82145bed7e8d8a8c11e0d2",
    "sha1_grams": [...],
    "entropy": 2.083932,
    "mimeType": "text/plain"
  },
  "file": {
    "name": "config.dat",
    "name_reverse": "tad.gifnoc",
    "path": "/system/config.dat",
    "createdOn": "2011-08-14T19:34:23Z",
    "modifiedOn": "2012-05-09T08:22:03Z"
  },
  "id": 26,
  "_type": ["file"],
  "_path": "1,a,n,",
  "_level": 3
}
```

The 256MB XIRAF XML test file used with Cassandra was loaded into an ElasticSearch index using the method described here. After flushing, the index was measured to be 4694MB, 18.3 times the size of the original data.

3.3.2 Query translation

The query interface that ElasticSearch offers by default is very rich, allowing a user to filter based on any conceivable property, including exact matches, value ranges and boolean constructions. As such, other than the translation from XIRAF's query language to a form that ElasticSearch understands. ElasticSearch makes a distinction between the concepts of a query, for which a score is calculated for each document in the result set, and that of a filter, which is not scored. Although both concepts are similar in their expressiveness and can be combined in the same ways, filters are more performant as ElasticSearch can cache the results of a filter, as opposed to a scored query. Requests for descendants of traces or other requests pertaining to the tree structure in the data set can be executed in the generic way described in Section 3.1.2.

Considering the example query in Listing 1.1, the combination of the three filter is translated to an ElasticSearch query expressed in its own query language. As scoring is currently not used, but the top level request to ElasticSearch is required to be query rather than a filter, a 'constant score query' is used, equipped with a boolean filter that expresses the example query in Listing 1.1:

Listing 3.3 ElasticSearch query to satisfy example query in Listing 1.1.

```
{"constant_score": {"filter":
  {"bool": {"must": [
    {"prefix": {"file.name_reverse": "fdp."}},
    {"numeric_range": {"content.size": {"from": 100}}},
    {"term": {"type": "deleted"}}
  ]}}
}}
```

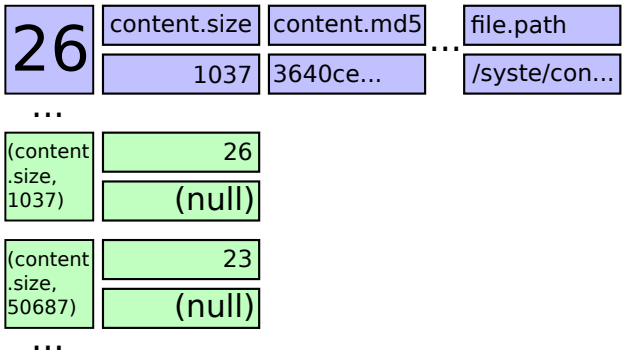

Figure 3.2 Data from the document in Listing A.1 as saved to HBase rows in the data and index tables.

3.4 HBase

Although HBase uses some structures similar to those used in Cassandra, the two differ in ways that make their approach to finding traces quite different. Figure 2.4 shows similar deficiencies in capabilities in comparison with Cassandra.

3.4.1 Data structures

Being modeled after Google's Bigtable, the data model used by HBase defines rows of data identified by a row key. A table is created that uses a trace identifier as the row key and a variable number of columns to store the properties and their values, all encoded as byte sequences as required by HBase. An index for all the different properties is created in a second table, for which the keys are composed of the combination of property names and values. All traces with the same value for a property are present in the same index row, where the column names are the trace identifiers of all traces that contain that specific combination of property and corresponding value. Figure 3.2 shows a part of the tables that are created for the document in Listing A.1. The top row show a row that stores the trace with identifier 26 as a single entity, storing properties like *"content.size"* with their corresponding values like with Cassandra. The other rows show index entries for the *"content.size"* property for two different values.

The requirement of supporting range queries is also supported by the index table, by executing a partial scan of the table. A range scan can be executed at the section of the index table ranging from the row key *(property, start value)* to the row key *(property, end value)*. As HBase treats row keys as byte arrays and orders these lexicographically, this approach hinges on the fact that rows are scanned in the

natural order for the second part of the compound row key, requiring the index entry for *(property, -1)* to occur before the index entry for *(property, 1)*. For string type values encoded in the UTF-8 standard this is the case as the byte sequence representing a UTF-8 string has the same lexicographical order as the natural order of the characters in the string itself [35]. For numerical values, this is non-trivial. By default, HBase will serialize integral number values in 2s-complement encoding. 2s-complement-encoded integral number values converted to byte arrays are not sorted in their natural order, sorting negative values after positive ones, for example. The same holds for IEEE-754-encoded floating point values. To overcome this, all numerical values have to be encoded in a non-standard way that keeps natural order when compared lexicographically. A detailed description of the encoding scheme can be found in Appendix B. Although examples for the encoding are provided using 8 bits for readability, the implementation of the encoding is used with 64-bit values.

The 256MB XIRAF XML file used with Cassandra and ElasticSearch was loaded into HBase tables using the data structures described above. The resulting tables totaled 7933MB in size after flushing, 31 times the size of the original data.

3.4.2 Query translation

As HBase supports a SCAN operation with a `Filter` object to determine whether or not to include a particular row, finding the result set for a query can be executed using a single SCAN operation on the data table, using a combination of `Filter` objects that together represent the user's query. Early experiments with filtered full table scans proved that the approach works, but the performance of it makes it unusable for a query interface as required by the provided context because the approach requires a full table scan for every query. The alternative is to use a custom made index table as described in Section 3.4.1 and depicted in Figure 3.2. By issuing a partial SCAN operation to the index table for each filter requested by the user, a number of intermediate results can be retrieved for each separate filter. As HBase does not offer a way to combine the results of these scans on the server side, they are processed on the client side.

The example query in Listing 1.1 can be processed with HBase using the following pseudo code procedure:

Listing 3.4 Pseudo code procedure to satisfy example query in Listing 1.1.

```
Results1 = ScanIndex(
        Key('file.name_reverse', 'fdp.'),
        Key('file.name_reverse, 'fdp.' + MAX))
Results2 = ScanIndex(
        Key('content.size', 100),
        Key('content.size', MAX))
Results3 = ScanIndex(
        Key('type', 'deleted'),
        Key('type', 'deleted' + MAX))

Results = Intersect(Results1, Results2, Results3)
```

The "Key" function creates a byte sequence usable as a key for the index table. The function "ScanIndex" is provided with start and end points and instructs HBase to start a SCAN operation at the specified point, retrieving identifiers of traces in between the start and end points. The value *MAX* rep-

resents the highest possible value a single byte can have and is used as an end marker, making sure HBase scans include all possible entries for the requested string type values, but nothing beyond. As with Cassandra, the intermediate result sets from separate filters for the example query need to be processed on the client side by means of intersection.

3.5 MongoDB

Like ElasticSearch, MongoDB offers a very rich query interface that allows the combination of a number of filters to find the traces that match those filters. Unsurprisingly then, Figure 2.5 displays an identical situation to that of ElasticSearch. Little effort outside the actions described in Section 3.1 were needed to be able to use MongoDB for the provided context.

3.5.1 Data structures

MongoDB stores collections of documents. As the relations among traces can be encoded using ancestor paths, each trace can be saved as a separate document. The documents saved to MongoDB are supplied as JSON-encoded version of traces identical to ElasticSearch. Listing 3.2 shows such a JSON document.

MongoDB natively allows searching documents within a collection, so no separate data structures have to be created. The data store does offer indexing techniques that speed up searches, which are enabled on properties that are expected to be used frequently in search requests. The maximum number of indexes allowed per collection is limited, however. For the version used in this work, the maximum number of indexes is 64. Although the XIRAF data model does not define 64 possible properties for traces, the addition of versions of properties that have been reversed and converted to lower case make the total number of properties in the documents that are saved to MongoDB larger than 64. Because of this, indexes have been specified on the properties that are used most often for queries—like *"_path"* and *"file.name_reverse"*—up to a total of 64 indexes. This leaves a number of properties unindexed, requiring a full table scan should they be used in queries. To facilitate easy balancing of the chunks of data used by MongoDB to balance data in the cluster, MongoDB's internal identifier property "`_id`" is as the *sharding key*.

The 256MB XIRAF XML test file used with to measure the resulting data set sizes was loaded in to a MongoDB collection and flushed to disk. The resulting collection occupied 3317MB on disk, 13 times the size of the original data set.

3.5.2 Query translation

MongoDB features a rich query interface based on providing a reference document that is used to evaluate whether a particular document in storage should be included in the result. Apart from specifying properties with required values, MongoDB's query interface allows more complicated expressions like string prefixes, numeric value ranges and logical operators.

All but the tree structure filters can be combined into a single reference document, reducing the

number of database requests required to answer a given query. As stated above, the tree structure expressions can be expressed as a regular filter if its inner query block is a filter containing only an identifier match, as in the query listed in Listing A.2.

Considering the example query in Listing 1.1, a reference document can be created, instructing MongoDB to find traces that match it. For requirements other than exact matches, MongoDB uses marker properties starting with a dollar sign. In the case of a string prefix requirement, `$regex` is used, accompanied by a simple regular expression to match strings that start with *"fdp."*. The numeric requirement for the *"content.size"* property is expressed using `$gte`, an acronym for *greater than or equal*, as required by the query. The three filters from the query are translated into the following reference document:

Listing 3.5 Reference document to satisfy example query in Listing 1.1.

```
{
  "file.name_reverse": {"$regex": "^fdp\\."},
  "content.size": {"$gte": 100},
  "type": "deleted"
}
```

3.6 Summary

Putting all solutions together in a single picture yields Figure 3.6. The index structures added to Cassandra and HBase as described in Sections 3.2 and 3.4 respectively are depicted on top of the functionality the data stores offer. To enable the solution as a whole to answer the type of queries required, a piece of query processing logic is added to the client side for both Cassandra and HBase. For Cassandra, this envelopes the slicing of the correct indexes as described in Section 3.2.2. The equivalent logic for HBase determines and processes the correct part of the index as described in Section 3.4.2. As mentioned in Section 3.1.2, the evaluation of tree structure queries using a non-trivial result set—the result of a query as opposed to a single trace identifier—as an evaluation context is not supported by any of the data stores, so a piece of query processing logic to accommodate for this is added to the stack for each of the four data stores.

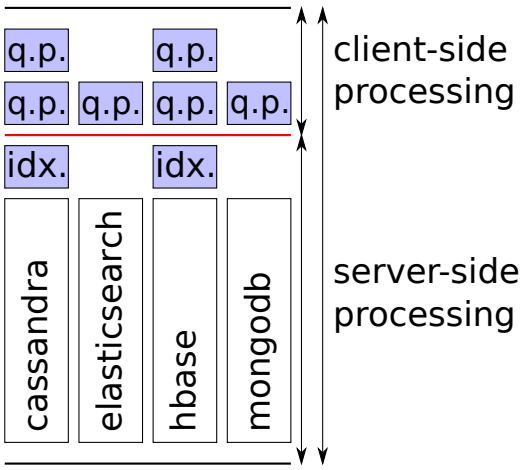

Figure 3.3 Components added to the data stores as described in Sections 3.2 through 3.5. Indexing structures are indicated with *idx.*, query processing logic is indicated with *q.p.*.

The client-server barrier has been depicted by a red line in Figure 3.6. As becomes clear from the figure, the amount of logic added to the complete software stack to satisfy the context's requirements is greater for the data stores that lacked sufficient index structures and/or query processing capa-

bilities by themselves. The logic added to the complete software stack here directly relates to the components that are reported to be lacking for the respective data stores in Section 2.

4 Experiment setup

Given experimental implementations of query execution engines for different data stores, experimental results allows us to provide an answer to the questions stated in Section 1.2. The details of these experiments are provided here. Descriptions of the experimental implementations of the query execution engines used to run the experiments are provided in Section 3.

4.1 Experiment scaling dimensions

As this research aims to find scaling characteristics for particular database deployments, the experiment needs one or more dimensions of scaling to provide us with a clear picture of the characteristics in question. Although the performance of a distributed system is often measured scaling the number of nodes that make up a cluster, there are more dimensions to be taken into account here. Measurements are recorded for each data store implementation, scaling the following variables:

Cluster size: The number of nodes that handle the data sets;
Data size: The size of the data set that queries are issued for;
Client load: The number of parallel clients that submit queries to the data stores.

The cluster size allows us to observe the effect of adding more nodes to a clustered setup on performance of the data store as a whole. This dimension can be related to the scalability requirement as stated in Section 1.1.1. Scaling the amount of data present in the data store allows us to see the performance implications of distributing query processing workloads for different data stores. This dimension is motivated from the observed and expected increase in data size of forensic investigations. Scaling the number of clients simulates a high load on the database, allowing us to determine how well it copes with many concurrent users as stated in the scalability requirement of Section 1.1.1.

4.2 Test data

The data used to execute the experiments is actual case data provided by the NFI. A number of data sets of various sizes have been selected from the available pool. The size of a data set is measured in the number of traces, the selections are made close to round numbers, as shown in Table 4.1. Furthermore, sample data was taken from all data sets serving as input for the query generation process described below.

Data set	*Number of traces*
set01	11549
set02	50814
set03	99158
set04	200411
set05	513952
set06	1099665
set07	1982229
set08	5118783
set09	10255300
set10	19979127

Table 4.1 Used data set sizes measured in number of traces.

4.3 Client simulation

4.3.1 Query generator

A query generator was built that is capable of creating XIRAF queries following the specification. Queries are generated using the probabilities encoded in an analysis result like that of Figure 1.1, making sure the total set of queries is closely related to the work load generated by real world use. By traversing the edges of the tree in the statistical model depicted by Figure 1.1, the query generator builds a query like it would occur in the application logs of XIRAF. Assigning queried values from sample data and miscellaneous settings like case sensitivity from statistical data associated with the context analysis described in Section 1.1.5.

The properties and corresponding values used by the queries are very dependant on the data they are to be evaluated on. To avoid generating queries that are not applicable to the test data set, both available properties and corresponding values for the generated queries are sampled from a target data set. Additionally, the query generator takes care to avoid queries that are 'nonsensical'. Taking random properties from the available set could generate queries combining properties from disjunct trace types, like searching for phone call logs that end with ".pdf". As such, the set of properties used to choose from in any step of the query generation process is reduced to a set that is compatible with properties already selected for previous steps. Should the reduction result in an empty set of properties, the generator abandons the current effort and starts the process for the current query over.

Some features available in the specification outlined in Section 1.1.4 have been disabled in the query generator, for various reasons. Keyword matches rely on a full text index of the content corresponding to a trace. As only ElasticSearch supports full text indexing natively, processing keyword match queries would rely on a call to an external full text index. Considering that the three other data stores would make an identical call at this point, performance metrics for the data stores would contain a constant factor where keyword match queries are concerned. Finding property value duplicates results in all traces that have the same value for a particular property that a previously selected trace has. This feature is used to find files with identical content between devices, for example. Due to the complexity of the feature and the fact that it is rarely used in comparison with other queries—occurring in 0.7% of all XIRAF queries analyzed for the statistical model supplied in Section 1.1.5—this feature is not supported for the implementations under investigation and as such never chosen by the query generator.

4.3.2 Client side configuration

Using data sampled from the data sets listed in Table 4.1, the query generator was used to generate 1000 queries for each data set. The generated query sets are issued to the database by a parallel benchmark runner program specially created for the experiments. The benchmark program issues the 1000 queries sequentially in a randomized order. To scale the client load as mentioned in Section 4.1, the benchmark runner is run on one to four machines, using multiple threads and parallel connections to scale beyond four machines. The effective number of parallel clients then becomes the total number of threads used to issue request to the data stores, each issuing the same 1000 queries in randomized order. Client load is scaled up using 1, 2, 3 and 4 client machines with a single thread and using 2, 4, 6, 8, 12, 16 and 32 threads on 4 machines, resulting in a total of eleven scaling points in the number of effective clients: 1, 2, 3, 4, 8, 16, 24, 32, 48, 64 and 128. The `pssh` program version 2.3.1 [50] was used to make sure the benchmark runners were started simultaneously on all client machines.

The vendors of all database products state that caching is vital for database performance. As such, to avoid recording timings for executions when caches are 'cold', an execution of the query set for the particular data set is run on a data store deployment before executing the experiments that are counted towards the results. The results of the executions done to warm caches are discarded. This approach gives the database software the chance to load the current working set into memory, providing a consistent result more accurate for real world use.

4.4 Cluster setup

Given the queries generated as in Section 4.3, benchmarks can be run from clients that issue queries to deployments of the selected data stores. The data store implementations are deployed on an eight node cluster, using an increasing amount of nodes to create a distributed setup. The number of machines used for the scaling factor are 1, 2, 4, 6 and 8. Additional machines are used as clients and extraneous roles required by the data store implementations.

Figure 4.1 depicts the network layout of the cluster that was used to run the experiments. The red nodes, labeled "D" are used to run the data store implementations under investigation. Blue nodes run extraneous roles needed by MongoDB and HBase. The first requires a config server to manage the cluster meta data, labeled "M". The latter requires an HBase master to manage the cluster meta data, labeled "B", an HDFS name node to manage the distributed file system HBase runs on, labeled "H" and a Zookeeper to coordinate distributed operations and client connections, labeled "Z". All of these extraneous roles are vital to the operation of the data store, but their influence in the performance of a cluster was observed to be negligible from resource monitoring software running on the cluster as the CPU load and network usage on the machines performing the roles were indistinguishable from when the machines were idle. Nodes running extraneous roles are not counted towards the cluster size of any particular data point. Green nodes, labeled "C", run client code, issuing queries using a predetermined number of threads to simulate more than four simultaneously running clients. Lastly, the gray squares labeled "S" represent network switches connecting nodes to the rest of the network. As shown, a total of two switches are involved in the cluster that is used. Hardware details of the machines used to run both client side code and data stores are available in Appendix C.

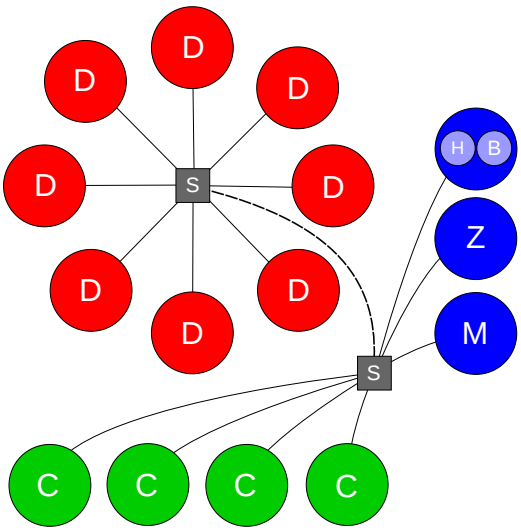

Figure 4.1 Network layout for the experiments carried out. "D": nodes used to run data stores, "C": nodes used as client machines, "Z": node running the Zookeeper, "M": node running the MongoDB config server, "H": the HDFS name node, "B": the HBase master, "S": switches. The HDFS name node and the HBase master were run on the same node.

4.5 Measurements

Measurements made by the data stores themselves on the server side are either hard to collect in a way that allows them to be related to a particular query or would incur a non-trivial logging overhead on the performance of the data stores. As such, the measurements made for our work are collected on the client side, ensuring that the timings are comparable among results for different data stores. Going back to Figure 3.6, arrows are added where timings were recorded for each query. The execution of a single query is timed from start to finish. Additionally, each round trip from the client side to the data store required for a single query is timed separately. As Cassandra and HBase require more round trips on separate filters, more round trip timings are collected for those data stores. In Section 5, we mention the timings collected this way.

5 Experimental results

The timings gathered from the various experiments that were carried out have been aggregated to be presented in a meaningful way. As described in Section 4.1, experiments were scaled in three dimensions: the amount of data in the working set for the data store, the number of nodes clustered together as a distributed data store and the number of parallel client applications issuing queries to the data store.

Although a total of 2200 possible data points could be gathered given the experiment setup—four different data stores, ten different data sets, five different number of nodes clustered together and eleven different number of parallel clients—the execution of the experiments have not yielded the full data set. Due to operational issues and combinations of data store, data set and number of nodes, a reduced total of 1497 data points have been created. Operational issues include the balancing operations for MongoDB, which required data sets to be re-inserted for each cluster size scaling point as MongoDB balances in a cautious manner without a means to speed this up. Re-inserting the data for each cluster size setup has saved a lot of time otherwise waiting for the balancer to complete, but has still proved a time consuming activity. Larger data sets caused issues with the implementation for Cassandra, requesting column slices larger than the maximum allowed size. Rather than change the implementation while executing the experiments, data points for Cassandra dealing with data sets above the size of a million traces have been omitted. HBase proved to be considerably slower in loading the data than other implementations. The reason for this was not discovered within the little amount of time available, causing us to omit data points for HBase dealing with data sets above the size of a million traces as well. Issuing a thousand queries per effective client per data point, a total of just under 44 million queries was issued in the execution of the experiments.

5.1 Execution times of query set

To avoid complex queries throwing off averages in the result data, query timings have been normalized. First, an average is taken for the timings for a specific query in a particular run. As queries can contain a different number of filters and tree structure expressions as described in Section 1.1.3, the average for the specific query is divided by the amount of operations undertaken for it—the number of filters and the number of tree structure expressions. This process yields a normalized timing for each query in the query set. Consider a query Q with three filters and four clients running in the experiment. All four clients will have executed query Q, resulting in four timings for the query. If the clients measured 210, 220, 230 and 240 milliseconds execution time for Q, the normalization is done by taking the average of the measurements—225—and dividing this by the number of filters in the query: 3. The normalized measurement for query Q is $\frac{225}{3} = 75$ milliseconds. Note that as each client executes the 1000 queries generated for a data set, the number of measurements that is averaged depends on the number of clients used. Ultimately, a thousand data points remain for each combination of scaling dimensions, representing the 1000 queries generated for a data set, each data point normalized for the amount of operation undertaken for it.

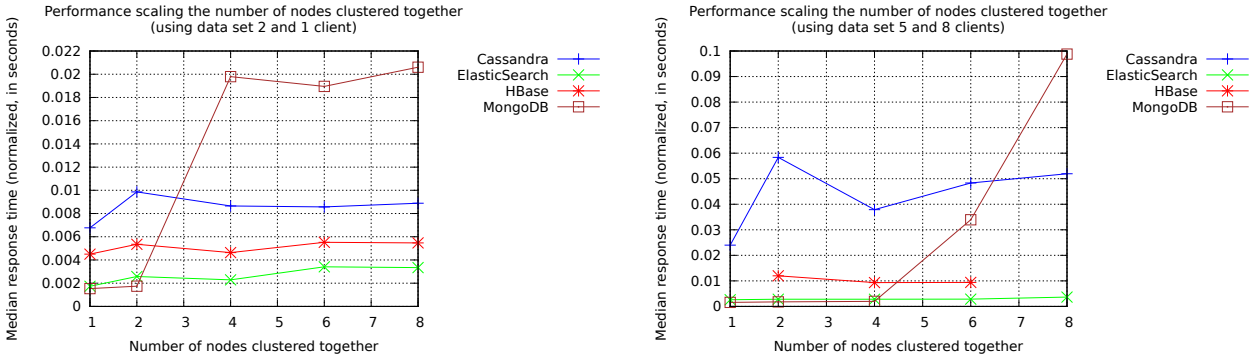

Figure 5.1 Median normalized query timings varying the number of nodes clustered together, in two different settings.

For the figures provided in this section, lower numbers on the y-axis represent faster response times and are considered better than higher numbers. As mentioned in Section 4.5, both the time to process a query as a whole and the steps required for the processing of each query have been recorded. The latter measurements have shown to be comparable to the normalized timings and have not been included separately in the figures provided here.

5.1.1 Scaling the number of nodes

Figure 5.1 shows the median performance measurements for the four data stores scaling the number of nodes clustered together. Like the scaling characteristics of the working set size, the number of nodes does not provide clear trends on all accounts. A notable observation is the trend that the mean query response time slowly goes *up* for ElasticSearch as more nodes are involved in processing queries. Both ElasticSearch and HBase seem to be quite constant in their response times, though don't show a clear trend.

5.1.2 Scaling the data set size

Figure 5.2 shows the median performance measurements for the four data stores scaling the number of traces in the working set. The data proves to be rather erratic, making it hard to make a statement on the scaling characteristics the data stores in the light of increasing working set sizes. Although MongoDB seems to present a near constant performance on two nodes using a single client, this behaviour is not observed for six nodes and eight clients. The reverse is observed for ElasticSearch, showing near constant performance using six nodes and eight clients while the behaviour on two nodes using a single client is more erratic.

5.1.3 Scaling the number of parallel clients

Figure 5.3 shows both mean and median query response times for the 99158 trace data set using two and six nodes. We make a number of observations based on the data shown.

Firstly, the difference between mean and median timings is large, pointing to a high spread in the query timings. This is observed in particular for MongoDB, showing a factor 44 difference between

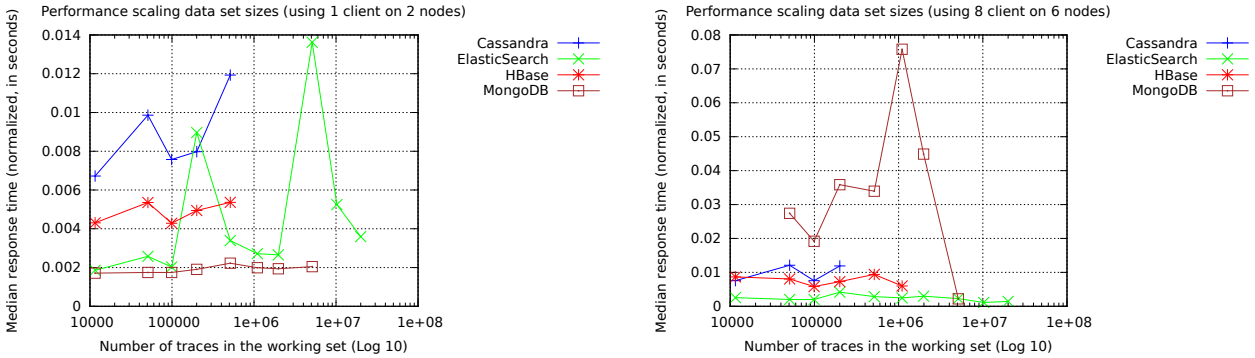

Figure 5.2 Median normalized query timings varying data set sizes in two different settings.

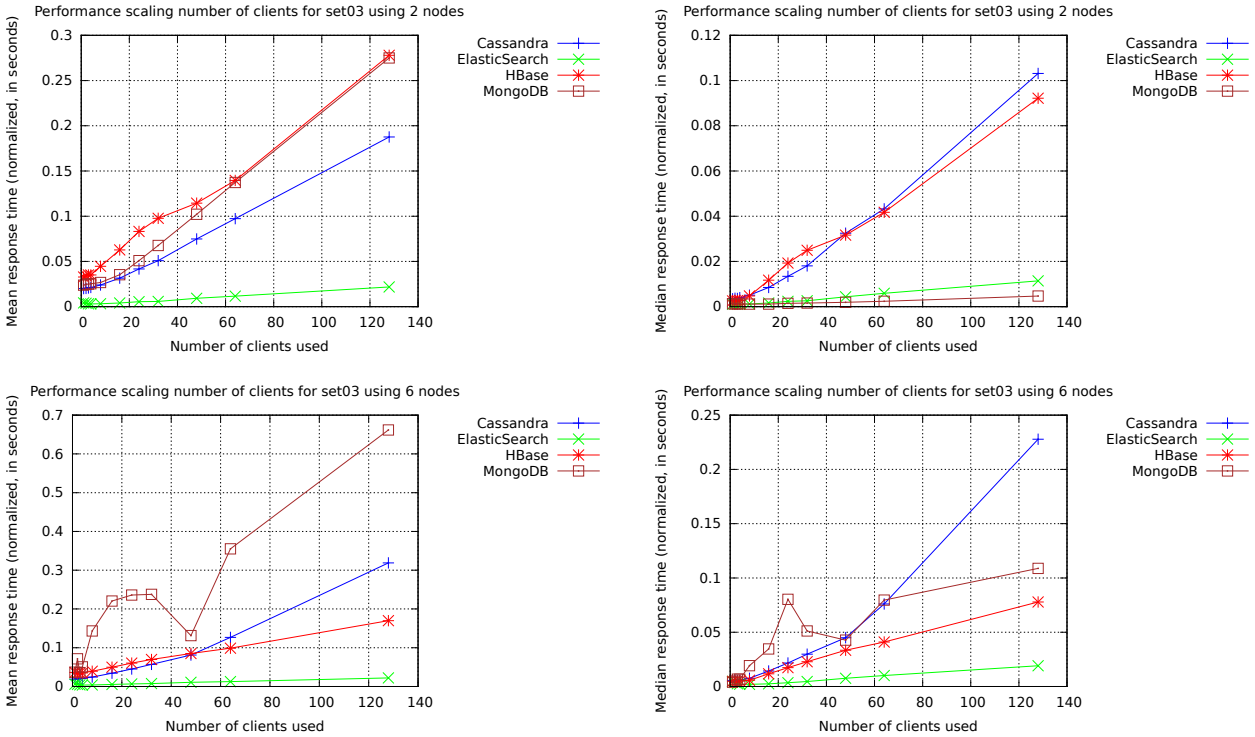

Figure 5.3 Mean and median normalized query timings varying the number of clients, using the 99158 trace data set on 2 and 6 nodes.

the mean and median values for the 99158 trace data set on 2 nodes. Using 6 nodes, the difference is smaller, though still considerable, showing a factor 6 difference. Other systems also display relatively large differences between mean and median timings, though are not considered to be as surprising as the timings for MongoDB. Only a few queries show a considerably higher response time for MongoDB. Upon closer inspection, these queries all contain a filter on a property that has not been indexed as MongoDB allows a maximum of 64 indexes per collection. Without an index present, MongoDB requires a full table scan to be able to process the query, resulting in a high response time. Secondly, the trend shown by most systems when the number of parallel clients is increased appears linear. Considering a minor overhead for each data point, linear trends can be interpolated through the origin point. Lastly, ElasticSearch is the most consistent in the overall picture, being considerably faster than the other systems in most situations.

5.2 Resource monitoring observation

The cluster of machines that was used to run experiments is fitted with machine monitoring software *Ganglia*, logging CPU and network resource usage among other things. Measuring performance, keeping an eye on various statistics has proven useful both in finding problems with implementations and corroborating trends identified in measurements from the benchmarks.

Firstly, network usage graphs showed limitations in both Cassandra and HBase as neither is capable of combining results of expressions. As this requires the intermediate result sets to be transported to a place where the combination is executed, Cassandra and HBase were observed to fully utilize the network interface of the server machines. Figure 5.4 shows the network utilization graph for Cassandra running experiments on the 50814 trace data set using four nodes. The graph shows an increase in network usage when more clients are used, reaching the hardware's maximum of just over a hundred megabytes per second. Whether the maximum link speed is reached at the client or server side depends on the number of server machines relative to the number of client machines of course. In either case, however, the performance of the solutions implemented for Cassandra and HBase can be bound by the maximum speed of the communication channels between the server(s) and the client(s). Neither ElasticSearch nor MongoDB have been observed to fully utilize network resources.

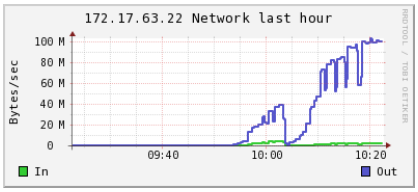

Figure 5.4 Network usage of Cassandra during experiments for the 50814 trace data set on four nodes.

CPU usage statistics showed another performance bound, particularly when running many clients in parallel. In cases where Cassandra and HBase were not bound by network resources, they were often bound by the amount of CPU time available, resulting in high Linux load numbers. For MongoDB, CPU utilization seemed to be related to the number of clients used, doubling resource utilization when the amount of clients was doubled, something the Linux load numbers seem to underline. At maximum CPU utilization, the execution time of a complete set of

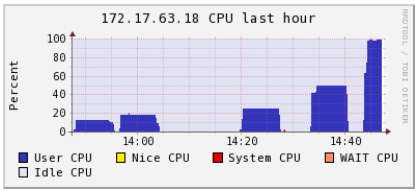

Figure 5.5 CPU usage of MongoDB during experiments for data set 2 on a single node.

queries doubled when the amount of clients was doubled. Figure 5.5 shows the CPU utilization graph for MongoDB running running experiments on data set 2 using a single node. The graph shows full CPU utilization at sixteen clients, doubling the resource allocation of the run with eight clients. ElasticSearch was able to serve more parallel clients than MongoDB before hitting full resource utilization, though it too reached full CPU utilization at high client counts.

6 Conclusions

The graphs presented in Section 5 provide us with data to make conclusions on the scaling characteristics of the data stores that were tested in this research. Considering research question 5, a linear trend has been observed considering the number of clients using the system simultaneously. For the other scaling dimensions—the size of the working set and the number of nodes clustered together—scaling characteristics cannot be clearly observed using the data from the experiments described in Section 4. Our work is considered inconclusive for research questions 3 and 4, though some interesting observations were made based on the available data. Although no clear trend can be confidently defined for the scaling of working set size and cluster size, a distributed search engine architecture is evidently well capable of performing in the provided context.

6.1 Context-specific evaluation

The setup as it was used to evaluate different data stores is in many ways equivalent to XIRAF, but differs significantly on a few points. The number of clients issuing database requests to XIRAF is quite a bit lower than the maximum amount used in the experiments described in Section 5. Scaling the number of clients does however provide an indication of the impact of parallelism on the performance of the application as a whole. The amount of data that is saved to the database but not being actively queried is higher for XIRAF than it has been in the data store deployments used for this research. It is assumed that the amount of inactive data has an insignificant impact on the performance of a data store, as is documented by the creators of the data stores used.

The features listed in Figure 2.1 have proven useful as a gauge on the implementation complexity of the database layer of an application like XIRAF. Considering the results as presented in Section 5, an answer to the second research question can be provided. The limitation on the number of indexes for MongoDB and the observed issues therewith aside, the two systems that provide both a rich query interface and integrated indexing capabilities perform better on the whole.

As stated in Section 4.3, full text queries have not been considered for the scope of our work. XIRAF ultimately needs this feature, requiring the NFI to add it to a system that does not natively support it. The fact that ElasticSearch—and MongoDB as of version 2.4—supports full text indexing out of the box is a great benefit, both lowering the implementation complexity and likely improving query response performance. Another valuable feature for XIRAF is *facetting*, which allows to obtain a statistical summary of the values for certain properties. The current user interface of XIRAF presents facets of data to its users, which currently require separate queries for each facet displayed. Enabling facets in combination with queries from the database layer removes a great number of queries that currently make up the set of requests from the application to the data store and will greatly improve performance of the application as a whole, while lowering the load on the data store cluster. All in all, the additional features offered by ElasticSearch combined with the steady performance in most of the data gathered by the experiments in Section 5, ElasticSearch is the best candidate to be ap-

plied to the provided context of the data stores that were evaluated here. The answer to the main research question, *which distributed data store architecture is the most suitable for the context of the NFI*, would be that of a distributed search engine such as ElasticSearch. Although ElasticSearch is not the only distributed search engine architecture available at the time of writing, the insights provided by our work shortens the list of candidates considerably. Considering the proof of concept implementations for the query execution engines and the success of ElasticSearch in the experiments, we conclude that the provided context benefits more from a search engine than a traditional database system.

6.2 General observations

Looking back at the component architectures and query execution solutions for all four data stores, there is a clear difference between ElasticSearch and MongoDB that provide indexing and flexible query interfaces and Cassandra and HBase, which required additional data structures to be applicable to our context. The descriptions of the query translations for the latter two data stores in Listings 3.1 and 3.4 are provided in a procedural pseudo code, whereas the descriptions for the former in Listings 3.3 and 3.5 show queries formatted in the domain-specific language for the respective data stores. In other words: where ElasticSearch and MongoDB are both able to express the example query in a declarative manner, allowing the data store to determine the most effective execution plan for the query, Cassandra and HBase require user effort to come to an execution plan to answer the query. We believe that this further illustrates that the implementation of the XIRAF data and query models in Cassandra and HBase are more fabricated on top of a data store designed for a different application domain than the implementations in ElasticSearch and MongoDB. The expressiveness of the queries for ElasticSearch and MongoDB thereby contributes to the conclusion for the second research question above.

The data set sizes measured for a 256MB XIRAF XML test file offer another interesting insight into the solutions created for our work. HBase taking considerably more space to save a data set than other solutions most likely allows for optimizations. Although this storage space 'explosion' would make HBase a poor choice for the implementation of the database component of XIRAF, we have not found the exact reason for the relative difference between HBase and the other data stores.

7 Discussion and future work

With this work, we've attempted to provide an overview and evaluation of the options for a next generation database component for a system like XIRAF. Although some interesting observations have been made, a number of points can be made regarding the work presented here.

7.1 Context particulars

The current user interface for XIRAF has a big impact on the queries issued to the database layer. The facetting mentioned in Section 6.1 is currently done with separate queries. Considering that facetting features have not been used for the proof of concept implementations described in Section 3, a database layer capable of creating facets alongside a query will dramatically change the statistical model provided in Section 1.1.5.

7.2 Operational experience

In implementing query execution engines on top of a number of different data stores and setting up and administering clustered configurations of the software, several additional remarks can be made on the products.

The query execution for Cassandra has been the most complex of the four to create. Although the Hector client library offers an extensive set of features that cover all of Cassandra's abilities, the API proven to be very verbose and cumbersome to work with. Additionally, the documentation of the library requires knowledge more knowledge of the inner workings of Cassandra than is to be expected of developers solely interested in the client side interaction with Cassandra. A similar experience goes for HBase, for which the pure byte sequence oriented API required lots of 'boiler plate' code to fabricate a working implementation. ElasticSearch and MongoDB were easier to work with in comparison.

A second notable piece of operational experience is the amount of effort required to set up a clustered deployment of MongoDB. The system requires an administrator to register each node in the cluster as a shard of the database. Although it is considered logical to make roles of nodes in a cluster explicit, for MongoDB there is no way to state this fact in static configuration. Additionally, MongoDB currently suffers from a maximum total cluster size of around 4PB due to an allocation of memory for each chunk in the config server that administers chunk placement. For our work, this is no issue as the data set sizes did not reach these numbers, though the limitation plays a role in the real world feasibility of MongoDB for the NFI.

7.3 Future work

The experimental results for the scaling benchmarks performed for this work make it apparent that benchmarking a complicated application context implemented on different database systems is a non-trivial task. From observations and experience in carrying out experiments described in Section 4 we formulate a number of areas for future work.

Synthetic data. The use of synthetic data would lower the chance of differences in data sets causing significant changes in results. Using real life case data has prevented the need for developing a test data generator, but small differences in the data sets that seem insignificant at first glance could have been the reason much of the experimental results have proven inconclusive.

Cluster size. Due to availability and existing configurations of hardware and software, experiments were limited to a total of eight nodes counted towards the database cluster sizes. Creating more scaling points in the number of nodes could provide a clearer picture on the scaling dimension that was presented in Section 5.

Specialized systems. Considering the success of ElasticSearch for the provided context, other systems that are more geared towards search than database operations could be interesting for deployment at the NFI. Not only the commonly known Lucene-based Solr [55] qualifies for this, but a system like HIndex [21] or Riak Search [42] could also prove interesting.

System tuning. Relatively little time has been invested into tuning the systems under investigation towards maximum performance. It is possible that particular bottlenecks observed with one of the systems used is more related to configuration than a more fundamental architectural trait. As such, conducting separate experiments with the configuration of the systems used could provide a better understanding of the forces at play.

7.4 Future developments

The versions of the software systems used for this work are all considered representative of the software, though development of these systems is an ongoing process. The Cassandra community in particular has voiced a number of improvement suggestions that could benefit its use for the context of this work. Using the Lucene library to facilitate secondary indexes will enable more flexible combinations of filters and sorting of results on top of the capabilities Cassandra currently has [49]. MongoDB version 2.4 facilitates a full text search feature, which would allow the indexing of the trace content alongside the meta data [40].

Bibliography

[1] Abadi, D., Madden, S., and Ferreira, M. Integrating compression and execution in column-oriented database systems. *Proceedings of the 2006 ACM SIGMOD international conference on Management of data - SIGMOD '06* (2006), 671.

[2] Abadi, D., Myers, D., DeWitt, D., and Madden, S. Materialization Strategies in a Column-Oriented DBMS. In *Proceedings of the 2007 international conference on data engineering* (2007).

[3] Alink, W. *XIRAF: An XML-IR Approach to Digital Forensics*. Msc. thesis, University of Twente, 2005.

[4] Aspnes, J., and Shah, G. Skip graphs. *ACM Transactions on Algorithms 3*, 4 (Nov. 2007).

[5] Bhoedjang, R. A. F., van Ballegooij, A. R., van Beek, H. M. A., van Schie, J. C., Dillema, F. W., van Baar, R. B., Ouwendijk, F. A., and Streppel, M. Engineering an online computer forensic service. *Digital Investigation 9*, 2 (2012), 96–108.

[6] Boag, S., Chamberlin, D., Fernández, M. F., Florescu, D., Robie, J., and Siméon, J. XQuery 1.0: An XML Query Language (Second Edition), 2010.

[7] Boncz, P., Grust, T., Van Keulen, M., Manegold, S., Rittinger, J., and Teubner, J. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (2006), pp. 479–490.

[8] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. Extensible Markup Language (XML) 1.0 (Fifth Edition), 2008.

[9] Bremer, J.-m., and Gertz, M. An Efficient XML Node Identification and Indexing Scheme, 2003.

[10] Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (2006), vol. 26.

[11] Clark, J., and DeRose, S. XML Path Language (XPath), 1999.

[12] Codd, E. F. Data Models in Database Management. In *ACM SIGMOD Record* (1980), pp. 112–114.

[13] Decandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of 21st ACM SIGOPS symposium on Operating systems principles* (2007), pp. 205–220.

[14] Feinberg, A. Project Voldemort: Reliable distributed storage. In *Proceedings of the 10th IEEE International Conference on Data Engineering* (2011).

[15] GOA, J., AND STEENKISTE, P. An adaptive protocol for efficient support of range queries in DHT-based systems. *Proceedings of the 12th IEEE International Conference on Network Protocols, 2004. ICNP 2004.* (2004), 239–250.

[16] GRUST, T. Accelerating XPath location steps. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data* (New York, New York, USA, 2002), ACM Press, pp. 109–120.

[17] JAGADISH, H., AL-KHALIFA, S., CHAPMAN, A., LAKSHMANAN, L., NIERMAN, A., PAPARIZOS, S., PATEL, J., SRIVASTAVA, D., WIWATWATTANA, N., WU, Y., AND YU, C. TIMBER: A native XML database. *The International Journal on Very Large Data Bases 11* (2002), 274–291.

[18] KIDO, K., AMAGASA, T., AND KITAGAWA, H. Processing XPath Queries in PC-Clusters Using XML Data Partitioning. *22nd International Conference on Data Engineering Workshops* (2006), 11–16.

[19] LAKSHMAN, A., AND MALIK, P. Cassandra - A Decentralized Structured Storage System. *Operating Systems Review 44*, 2 (2010), 35–40.

[20] LI, C., LING, T. W., AND HU, M. Efficient updates in dynamic XML data: from binary string to quaternary string. *The VLDB Journal 17*, 3 (Sept. 2006), 573–601.

[21] LI, N., RAO, J., SHEKITA, E., AND TATA, S. Leveraging a scalable row store to build a distributed text index. In *Proceeding of the first international workshop on Cloud data management* (New York, New York, USA, 2009), ACM Press, p. 29.

[22] MALEWICZ, G., AUSTERN, M. H., BIK, A. J. C., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 International Conference on Management of Data* (2010), pp. 135–145.

[23] MEIER, W. eXist: An Open Source Native XML Database. *Web Services and Database Systems 2593* (2003), 169–183.

[24] MOORE, G. E. Cramming more components onto integrated circuits. *Electronics 38*, 8 (1965).

[25] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The log-structured merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996), 351—-385.

[26] O'NEIL, P., O'NEIL, E., PAL, S., CSERI, I., SCHALLER, G., AND WESTBURY, N. ORDPATHs: Insert-Friendly XML Node Labels.

[27] PAGNAMENTA, F. *Design and Initial Implementation of a Distributed XML Database.* Msc. thesis, University of Dublin, 2005.

[28] RABL, T., SADOGHI, M., JACOBSEN, H.-A., GÓMEZ-VILLAMOR, S., MUNTÉS-MULERO, V., AND MANKOVSKII, S. Solving Big Data Challenges for Enterprise Application Performance Management. In *Proceedings of the VLDB Endowment* (2012), vol. 5, pp. 1724–1735.

[29] RAMABHADRAN, S., HELLERSTEIN, J., RATNASAMY, S., AND SHENKER, S. Prefix Hash Tree, An Indexing Data Structure over Distributed Hash Tables. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing* (2004).

[30] RATNASAMY, S., FRANCIS, P., HANDLEY, M., SHENKER, S., AND KARP, R. A Scalable Content-Addressable Network. In *Proceeding of the 2001 SIGCOMM conference* (2001), pp. 161—-172.

[31] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference* (2001), pp. 329–350.

[32] SCHMIDT, C., AND PARASHAR, M. Squid: Enabling search in DHT-based systems. *Journal of Parallel and Distributed Computing 68*, 7 (July 2008), 962–975.

[33] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A Scalable Peer-to-peer Lookup Service for Internet. In *Proceedings of the 2001 SIGCOMM Conference* (2001), pp. 149–160.

[34] VAN RENESSE, R., DUMITRIU, D., GOUGH, V., AND THOMAS, C. Efficient reconciliation and flow control for anti-entropy protocols. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware* (New York, New York, USA, 2008), ACM Press.

[35] YERGEAU, F. RFC 3629: UTF-8, a transformation format of ISO 10646, 2003.

[36] ZHAO, B. Y., KUBIATOWICZ, J., AND JOSEPH, A. D. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. *Computer Science*, April (2001).

[37] ZHOU, J., AND ROSS, K. A. Buffering Database Operations for Enhanced Instruction Cache Performance. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data* (2004).

[38] ZUKOWSKI, M., BONCZ, P., AND NES, N. MonetDB/X100 - A DBMS In The CPU Cache. *Bulletin of the Technical Committee on Data Engineering 28*, 2 (2005), 17–22.

Secondary sources

[39] 10GEN, INC. Mongodb. http://www.mongodb.org/, 2013. Visited: March 28, 2013.

[40] 10GEN, INC. Text search — mongodb manual 2.4.1. http://docs.mongodb.org/manual/core/text-search/, Apr. 2013. Visited: April 12, 2013.

[41] BASHO TECHNOLOGIES, INC. http://basho.com/riak/, 2013. Visited: April 14, 2013.

[42] BASHO TECHNOLOGIES, INC. Riak docs — searching. http://docs.basho.com/riak/1.3.0/tutorials/querying/Riak-Search/, Jan. 2013. Visited: March 27, 2013.

[43] DALTON, M., AND DIMIDUK, N. Orderly. https://github.com/ndimiduk/orderly, Apr. 2011. Visited: April 12, 2013.

[44] DATASTAX. About data partitioning in cassandra | datastax cassandra 1.1 documentation. http://www.datastax.com/docs/1.1/cluster_architecture/partitioning, 2013. Visited: April 14, 2013.

[45] DATASTAX. About indexes in cassandra | datastax cassandra 1.1 documentation. http://www.datastax.com/docs/1.1/ddl/indexes, 2013. Visited: April 16, 2013.

[46] ECHAGUE, P., AND MCCALL, N. Hector — java client for cassandra. http://hector-client.org/, 2011. Visited: April 14, 2013.

[47] ELASTICSEARCH BV. Overview — elasticsearch. http://www.elasticsearch.org/overview/, 2013. Visited: March 28, 2013.

[48] HYPERTABLE INC. Architecture | hypertable — big data. big performance. http://hypertable.com/documentation/architecture/, 2012. Visited: April 14, 2013.

[49] LUCIANI, J. [#cassandra-2915] lucene based secondary indexes. https://issues.apache.org/jira/browse/CASSANDRA-2915, July 2011. Visited: April 12, 2013.

[50] MCNABB, A., AND CHUN, B. N. parallel-ssh — pssh: Parallel ssh tools. http://code.google.com/p/parallel-ssh/, Feb. 2012. Visited: April 12, 2013.

[51] NEO TECHNOLOGY, INC. Neo4j, the graph database — learn, develop, participate. http://www.neo4j.org/, 2013. Visited: April 14, 2013.

[52] ORACLE. Guide to Scaling Web Databases with MySQL Cluster. White paper, May 2012.

[53] THE APACHE SOFTWARE FOUNDATION. Apache lucene — index file formats. http://lucene.apache.org/core/3_6_2/fileformats.html#Document%20Numbers, Dec. 2012. Visisted: April 11, 2013.

[54] THE APACHE SOFTWARE FOUNDATION. Apache lucene core. http://lucene.apache.org/core/, 2012. Visited: March 28, 2013.

[55] THE APACHE SOFTWARE FOUNDATION. Apache solr. http://lucene.apache.org/solr/, 2012. Visited: March 28, 2013.

[56] THE APACHE SOFTWARE FOUNDATION. Edgengramtokenfilter (lucene 3.6.2 api). http://lucene.apache.org/core/3_6_2/api/all/org/apache/lucene/analysis/ngram/EdgeNGramTokenFilter.html, Dec. 2012. Visited: April 15, 2013.

[57] THE APACHE SOFTWARE FOUNDATION. Apache hbase — home. http://hbase.apache.org/, Mar. 2013. Visited: March 28, 2013.

[58] THE APACHE SOFTWARE FOUNDATION. Hdfs architecture guide. http://hadoop.apache.org/docs/stable/hdfs_design.html, Feb. 2013. Visited: March 27, 2013.

[59] TINKERPOP. Gremlin — tinkerpop. http://gremlin.tinkerpop.com/, 2009. Visited: April 14, 2013.

A XIRAF data and query examples

Both analysis results and XIRAF queries are expressed in XML. The following provides examples of the data and query format used by XIRAF.

A.1 Data format

The formal specification of the data format for XIRAF is expressed in an XML Schema. Though useful for automated consumption of such a specification, we believe an example of the data format will provide a better insight into the data model than the schema. Listing A.1 provides a simple folder as one might find on a computer's file system with a number of sub folders and files.

Listing A.1 Example XIRAF XML document.

```
<item id="1">
  <item id="10">
    <item id="23">
      <content>
        <size>50687</size>
        <md5>eaa0290aeee8ebe5924f204fc3c8b934</md5>
        <sha1>17ddf2ec050c2109bc0e7a9390ae35ec31157add</sha1>
        <entropy>4.20320</entropy>
        <mimeType>image/jpeg</mimeType>
      </content>
      <file>
        <name>profile.jpg</name>
        <path>/system/user/profile.jpg</path>
        <createdOn>2012-02-27T23:15:40Z</createdOn>
      </file>
      <picture>
        <width>300</width>
        <height>400</height>
      </picture>
    </item>
    <folder>
      <name>user</name>
      <path>/system/user</path>
      <createdOn>2011-08-14T19:35:56Z</createdOn>
    </folder>
  </item>
  <item id="26">
    <content>
      <size>1037</size>
      <md5>3640ce36ca049c9cd81cf354cdfe5c76</md5>
      <sha1>4c5f05440b87dcf86e82145bed7e8d8a8c11e0d2</sha1>
      <entropy>2.083932</entropy>
      <mimeType>text/plain</mimeType>
    </content>
    <file>
      <name>config.dat</name>
      <path>/system/config.dat</name>
      <createdOn>2011-08-14T19:34:23Z</createdOn>
```

```
      <modifiedOn>2012-05-09T08:22:03Z</modifiedOn>
    </file>
  </item>
  <folder>
    <name>system</name>
    <path>/system</name>
    <createdOn>2011-08-14T19:30:01Z</createdOn>
  </folder>
</item>
```

The document shows a total of four traces, expressed as `<item>` elements in XML. The top level folder of the file system is named *"system"*, which contains a file named *"config.dat"* and a folder named *"user"*. Note that child traces are provided before the information on their parent. Properties of a trace are encoded as child *elements* of an element that encodes the trace type they are attached to. So, the property that encodes the file name of a trace is encoded as a `<name>` XML element enclosed by a `<file>` XML element. The example shows one exception to this rule, as 'content' is not defined to be a trace type by the data model, but rather a collection of properties assigned to traces that have byte content associated with them. Both the configuration file and the profile picture in the example have byte content associated with them, the folders do not. Four different property value types can be distinguished: integral number, string, decimal number and time stamp. These represent, for example, a file's content size, a file's name, a file's entropy and a file's creation time respectively.

A.2 Query language

The formal specification of the query language for XIRAF is expressed in an XML Schema like the data format. Listing 1.1 in Section 1.1.4 lists a XIRAF query containing three filters on the data set. Below, a number of additional queries are provided that showcase more complex features of the query language.

Listing A.2 Example XIRAF query with a tree structure expression.

```
<has-type invert="true">
  <type>picture</type>
  <descendants>
    <has-id>
      <id>9283</id>
      <unrestricted-document />
    </has-id>
  </descendants>
</has-type>
```

Listing A.2 shows a query with two filters and a tree structure expression. The innermost filter simply requires the traces in its result set to have trace identifier 9283. As trace identifiers are required to be unique, the filter will have either one or zero results. A tree structure expression is applied to the singular result, retrieving all descendants of the trace with identifier 9283. A final filter is applied to the result set of the tree structure expression, leaving only traces that do *not* have trace type picture. This query could for example be issued to request the traces contained in a particular file archive—with identifier 9283—leaving out pictures.

Listing A.3 Example XIRAF query with a logical disjunction (union) and tree structure expression.

```
<union>
  <children>
    <has-text-property>
      <properties><property>
        <type>file</type>
        <property-name>extension</property-name>
      </property></properties>
      <case-sensitive>false</case-sensitive>
      <match>exact</match>
      <value>zip</value>
      <unrestricted-document />
    </has-text-property>
  </children>
  <has-type>
    <type>email</type>
    <unrestricted-document />
  </has-type>
</union>
```

Listing A.3 shows a query with two filters, a tree structure expression and a union. The first operand of the union is a request for all child traces of files whose extension is equal to *"zip"*, allowing different casing, like *"ZIP"*. The second operand of the union is a request for all traces of type email. The result of the query as a whole is a result set with both email type traces and traces that are located in zip files.

B Lexicographically sortable number encoding

From a lexicographical point of view, the lowest possible value for a byte is `00000000`. Consequently, the highest value is `11111111`. In most systems, integral numbers are encoded using 2s-complement and decimal numbers using IEEE-754. Although the CPU of a system will be able to compare two number encoded with either 2s-complement or IEEE-754, neither byte sequences conform to the natural order of the numbers they encode. In an environment that is only capable of comparing byte sequences—which is the case for HBase in our work—an alternate encoding scheme is needed to ensure the result of an operation conforms to the natural order of the encoded numbers. The number encoding schemes presented here are inspired by "Orderly", a library aimed at adding typed value support to HBase [43].

B.1 Integral numbers

The most significant distinction in numerical value is the sign of the value: negative values should be sorted before positive values. By making sure to explicitly sign the encoded value, the distinction can be made explicit. Marking the most significant bit `1` for positive values and `0` for negative values satisfies the natural order. Considering a single byte as an example, 7 bits are left for encoding the numerical value after signing. Using the absolute value of the number to be encoded, a bit sequence representing a positive number can be obtained by using a regular binary encoding. As such, the number 2 would be encoded as `10000010`. For positive values this encoding guarantees an equal natural and lexicographical ordering, while negative values would be sorted before positive values, but in opposite order among among themselves. To counteract the last issue, all bits but the sign bit are inverted for negative values, reversing the sort order to the natural order. The introduction of the signing bit and inversion of negative numbers satisfies the requirement of aligning natural and lexicographical sort order, at the cost of not being able to encode the lower bound of the 2s-complement encoding scheme. Table B.1 shows a number of integral numbers encoded in both sortable and 2s-complement encodings.

Value	Sortable	2s-complement
-128	*(not possible)*	1 0 0 0 0 0 0 0
-127	0 0 0 0 0 0 0 0	1 0 0 0 0 0 0 1
-1	0 1 1 1 1 1 1 0	1 1 1 1 1 1 1 1
0	1 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
1	1 0 0 0 0 0 0 1	0 0 0 0 0 0 0 1
127	1 1 1 1 1 1 1 1	0 1 1 1 1 1 1 1

Table B.1 Various integer values encoded as a single byte in both lexicographically sortable and 2s-complement encodings.

Value	Sortable	IEEE-754-style
-1.6	0 0 1 1 0 1 1 1	1 1 0 0 1 0 0 0
-1.0	0 1 1 1 1 1 1 0	1 0 0 0 0 0 0 1
-0.0	0 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0
0.0	1 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
1.0	1 0 0 0 0 0 0 1	0 0 0 0 0 0 0 1
1.6	1 1 0 0 1 0 0 0	0 1 0 0 1 0 0 0

Table B.2 Various floating point values encoded as a single byte in both lexicographically sortable and IEEE-754-style encodings using one sign bit, two exponent bits and five significand bits.

B.2 Decimal numbers

A similar approach is used for floating point values. As IEEE-754-encoded numbers already have a sign bit, the first distinction is made by flipping the most significant bit, as the IEEE-754-standard uses 1 as an indication of a negative value. As with integer numbers, all bits but the sign bit are inverted for negative values to ensure the lexicographical ordering is equal to the natural ordering of floating point numbers. Table B.2 shows a number of decimal values encoded in both sortable and IEEE-754 encoding.

C Cluster details

The experiments described in Section 4 were run on a cluster of identical machines. The hardware of these machines is listed below.

Component	*Details*
CPU	2 × Intel Xeon E5620
Total CPU cores	8 physical, 16 logical
Main memory	64GB
Disk setup	3 × 1 TB disks, LVM, XFS
Network link speed	1 Gbit