XML and NoSQL DBMS: Migration and Benchmarking

Author:
Prakash Thapa

University of Konstanz

April 9, 2015

Abstract

As dynamic data is growing rapidly, database vendors search for the alternative of the classical database management system such as RDBMS. XML and NoSQL databases are two non-relational database management systems growing since the popularity of Big Data. This thesis focus on the comparative analysis of these two new database system based on use cases and existing solutions. On the first part, our focus will be data migration from one system to another. We will also discuss the performance of both systems based on standard queries.

....

Zusammenfassung(German Abstract)

XML und NoSQL

Acknowledgments

.....

Contents

Chapter 1

Introduction

1.1 Motivation

As the digital world growing very fast, the massive amount of data collected today in the field varying from business to scientific research, is becoming complex for storage, querying and providing mechanism for failover. As the data collection grows so largely, the traditional data management tools e.g. relational database management systems(RDBMS) are struggling to handle such data effectively. The pre-design rigid schema structure of RDBMS made more complex for variety of data. High data velocity where massive read and write operations possible from different geographic locations, storage of structured, semi-structured and unstructured data in gigantic volume, together is refered as *Big Data*. This has become a global phenomena which added more complexities on RDMS.

New database technologies NoSQL and XML Databases came into existence to overcome above mentioned problems of RDBMS. These systems are not just able to solve the issues of existing data management systems, but also rise as alternative in their way. Both these new technologies focus on varieties of data in large volume, dynamic schema and scalability.

There are some research analysis between NoSQL and RDBMS. Nance et al. (2013) examine the pro and cons of both NoSQL and RDBMS, Cattell (2011) analyze similarities between scalable SQL and NoSQL databases, Hadjigeorgiou et al. (2013) compare the performance between MySQL cluster as RDBMS and Mongodb as NoSQL Database. There are also some research related to RDBMS with XML or XML Database(Jiang et al. (2002), Shanmugasundaram et al. (1999)). But there are not much more research in the field of NoSQL and XML database together. So this thesis focus on these two new database systems. Migration of Data from XML database to NoSQL as well as performance of both system based on standard query will be examined.

1.2 Contribution

The main contribution of this thesis is that it provides the necessary techniques and algorithms for migrating data from an XML database to NoSQL databases. More specifically, it will focus on databases like MongoDB, Couchbase and Rethinkdb as NoSQL databases and BaseX as XML Database. To approach this challenge, it is first necessary to understand the general architecture and data model of each of these databases as well as the way how they are queried.The performance com-

parison of these two systems will be based XMARK dataset and its 20 standard queries (Schmidt et al., 2002). These 20 queries of XMARK project will be translated to each of NoSQL databases.

1.3 Scope of Thesis

....

1.4 Overview

This thesis is divided into three main sections. Chapter 2, we will discuss the two new databases system and the scope of thesis. Chapter 3 defines the techniques and necessary algorithms to convert XML to JSON. In same section, the XMARK dataset will be introduced and procedure to store in NoSQL databases. In Chapter 4, focuses on the performance tests and comparative analysis of each of the NoSQL databases with BaseX, based on the XMark benchmarking project.

Chapter 2

NoSQL and XML Databases

2.1 XML Database

XML Database or sometime written also Native XML Databases (NXDs) are document oriented databases that store XML data and all components of XML based on a specified logical model. This model might differ to individual NXDs, but all databases include at least elements, attributes, PCDATA, and document order. An XML document is the fundamental unit of storage, so each document should be in a valid XML format. XQuery 1.0 and XQuery 3.0 are standards query languages recommended by W3C for NXDs. XQuery also includes XPath as sub-language. In addition to XQuery and XPath, XML databases also support XSLT, a method of transferring XML to other documents like HTML, plain text and XSL Formatting Objects.

2.1.1 BaseX

BaseX is a Native XML database management system and XQuery Processor. It uses tabular representation of XML data tree to store XML document (Grün, 2010). In this dissertation, BaseX is represented as XML database. ....

2.2 NoSQL Database

NoSQL database "Not Only SQL" is distributed data management system for large and variable data. The main focus of any NoSQL database are scalability, performance and availability (Hecht and Jablonski, 2011). Based on the Data model and design architecture, NoSQL databases are categorized into 4 groups:

- **Key/Value store** The Data representation of key-value stores are based on attribute pair, the data model is expressed as collection of $<key,value>$ tuples. The key is unique and value can be varied. The query operation on data can be uniquely performed by a key. There are no alternative ways to access or modify data without key. DynamoDB, Riak, Redis are some examples of key-value stores.

- **Document oriented databases** are based on semi-structured data model, where unique key stores a value that contains a tree like structure called *document*. It is enclosed with key-value pair in JSON or in JSON like data format (Hecht and Jablonski, 2011). In document store database, data can be

access by using key or specific value. Some of the examples of document store databases are Mongodb, Couchbase, RethinkDB, CouchDB etc.

- **Column family** is also known as wide-column database stores data tables as section of a column. Cassandra, BigTable, HBase are categories as wide column database.

- **Graph databases** are based on graph theory. In these databases, data are represented as a graph in the form of nodes and edges similar to social network. They store entities and relationship between these entities. Neo4J and OrientDB are two example of these databases.

As there are many categories and varieties of database system, our focus in this thesis is limited to document oriented databases of NoSQL databases.

2.2.1 MongoDB

MongoDB is a schemaless document oriented database developed by MongoDB Inc.(then 10gen Inc.) and Open Source Community. It is intended to be a flexible, fast and Multi-datacenter scalable system.

Data design MongoDB stores data as documents. Beside storing a document using unique key, MongoDB allows to query by every attributes of document. This made possible to index every field of a document. Documents with similar schema are group together in a collection. A database contains one or more collections As MongoDB is schema-free system, a collection may contain any type of documents but mostly it has documents with similar schema. Collections do not enforce document to structure data rather requirements of the our application. Documents are modeled as JSON format whichN is a binary variant of JSON supports additional data types like ObjectId, timestamp, datetime etc. In MongoDB, there are two principles that allow application to represent documents and their relationship: *reference* and *embedded* documents.

Embedded Embedded documents captures relationships between the data by storing related data in a single document structure. The documents in this method are structured as sub-documents in the form of Array or/and Object (Gao).

Reference MongoDB has no support for joins, so, related data are stored in a single document. In some cases, it makes sense to store related data in separate documents, typically in different collection or even database. Reference stores the relationships between data by including links and references from one document to another as in Figure 2.1a. These references can be created in two way.

- **Manual reference** The application handle the relationship between the related data in manual reference. The *_id* field of one document is referenced to another document.

- Database reference(*DBRefs*)

The application can resolve these reference to access the related data. The normalize data model

```
{
        title :" MongoDB ",
        last_editor :"172.5.123.91" ,
        last_modified :new Date ("9/01/2015") ,
        body :" MongoDB is a..." ,
        categories :[" Database ", " NoSQL ", "
           Document Database "] ,
        reviewed :false
}
```

Code 2.1: MongoDB sample document

Document A document is a abstraction and storable unit in MongoDB. It is the data structure represented in the form of JSON.

Data-types for Document Fields MongoDB support following datatypes

- scalar types: **boolean**, **integer**, **double**
- character sequence type: **string**(encoded in UTF-8), **regular expression**
- **Object**: All BSON-Objects
- **ObjectId** : It is BSON type 12 byte long binary value to uniquely identify the document. Documents are stored in a collection with _id field which also act as primary key with the value of ObjectId. The object id datatype is composed of the following components:
  - timestamp in seconds since epoch (first 4 bytes)
  - id of the machine assigning the object id value (next 3 bytes)
  - id of the MongoDB process (next 2 bytes)
  - counter (last 3 bytes)
- **null**
- **array**
- **date**

Indexing Each document in MongoDB is uniquely identified by a field *_id* which is a primary index. Hence, the collection is sorted by *_id* (Gao). In addition to primary index, MongoDB provides the mechanism to create secondary indexes for all document fields.It supports various user defined indexes for field values of documents including single field index, multikey index, multidimensional index, geo-spatial index, text index and hash index. Single field, multidimensional and multikey index are organized using B-tree, whereas geospatial index is implemented using quad trees.

- *Single field index* only includes data from a single field of documents in a collection.

(a) Reference document (b) Embedded document

Figure 2.1: MongoDB document structure

- *Compound index* holds reference to the multiple fields within a collection's documents.
- To index a field that contains an array value, MongoDB provides special indexing called *Multikey index*.
- *Text index* supports efficient search of string content in documents.

Query Model Queries in MongoDB are expressed in a JSON like syntax and send to MongoDB as BSON objects by database drivers(Orend, 2010). MongoDB's query can be implemented over all documents inside collections, including embedded object and arrays. The Query model support following features:

1. Queries over documents, embedded subdocuments and arrays
2. Comparison operators
3. Conditional Operators
4. Logical Operators: AND and OR
5. Sorting
6. Group by
7. Aggregation per query

In addition to this MongoDB provide a features to for complex aggregation with the use Of MapReduce. The result of MapReduce either can be store as a collection or be removed after result has been return to clientOrend (2010).

System Architecture MongoDB can be run in two mode. In stand-alone mode a single *mongod* demon is running in a single node without any distribution. The other mode is sharded mode, where various services of MongoDB are distributed to several nodes.

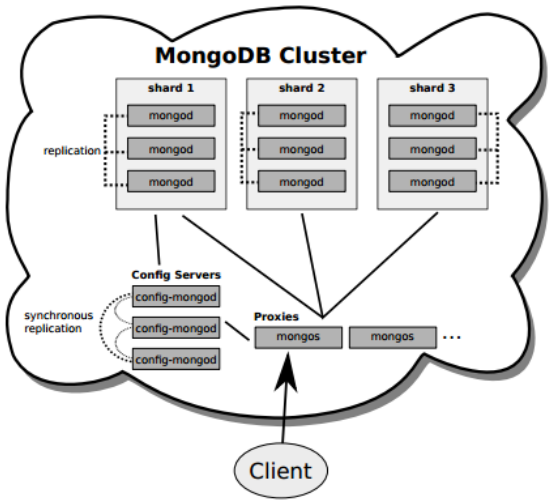


Figure 2.2: MongoDB Clusters

2.2.2 Couchbase Server

Couchbase Server is designed to be an operational data store which means it is solution for real-time data access.It is NoSQL database that facilitates both as a key-value store as well as document store.It As key-value store, it is able to store multiple data type such as strings, numbers, datetime, and booleans as well as arbitrary binary data. The key-value generally treated as opaque Binary Large Object(BLOB) and do not try to parse it at the time of query. For document store, data need to be store in the valid JSON format. Data in Couhbase Server are stored in logical unit called Buckets. Buckets are isolated to each other which also have their own RAM quota and replica settings. These buckets can be technically compared as `database` in Mongodb or other RDBMS. Couchbase recommends as few as possible number of buckets even it is possible to have up to ten buckets in a single cluster.These buckets theoretically contain any type of data. All data type other than JSON can be retrieved only by their key. So, it is important to check meta type of data stored in a single document before retrieval.

Metadata For every value stored in database, Couchbase Server generate following meta information that is associated with the document (Ostrovsky and Rodenski, 2014, p. 26).

- Expiration The Time to Live(TTL) also named expiration time is the life time of the document. By default it is 0 which indicates it will never expire, also can be set as Unix epoch time after which document is removed.

- Check-and-Set(CAS) The CAS value is 64-bit integer that is updated by server when associated item is modified. It enables to update information only if unique identifier matches the identifier of the document that need to be update. CAS is used to manage concurrency when multiple client attempts to update the same document simultaneously.

- Flags Flags are as 32 bit integer and are set of SDK specific use for SDK specific need. For example, format in which data is serialize or data type of the object being stored.

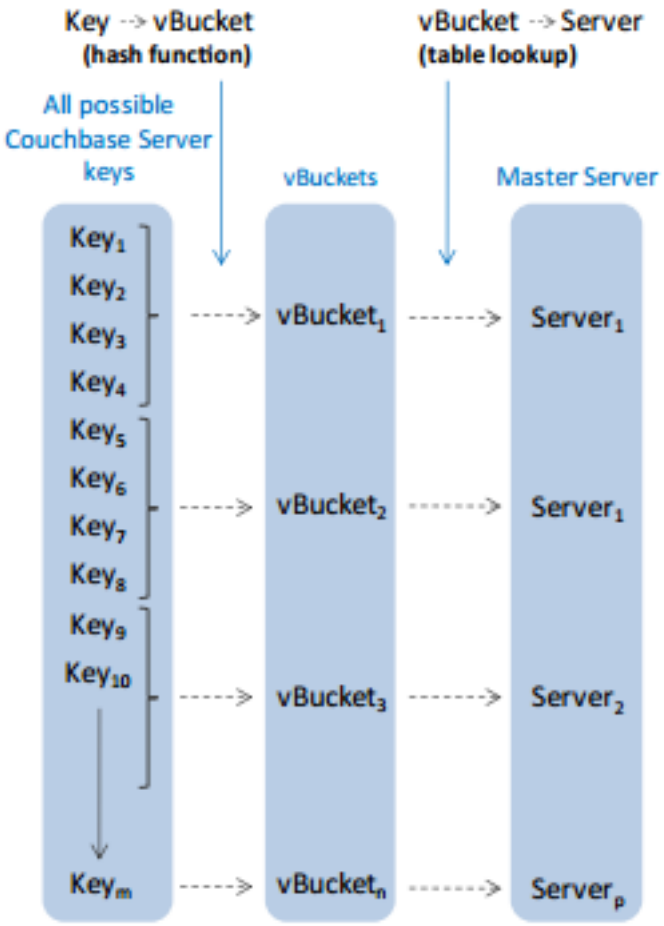


Figure 2.3: Couchbase Server vBucket

In addition of expiration, CAS and flags, three other meta information are stored at the time of document creation. The document's key *id* is saved as part of metadata. *type* is the type of document either **json** for all valid JSON documents and Base64 encoded string for all other than JSON.

Document key Every value in Couchbase Server is saved in unique key called document key. Unlink MongoDB, Couchbase do not generate document key automatically, it should be created manually a string value upto 250 characters.

Bucket and vBucket Couchbase Server uses data bucket as a logical container of information. It provides a logical grouping of physical resources within a cluster (Lichtenberg and Ward, 2013). A bucket is equivalent to a database in MongoDB or RDBMS. unlike MongoDB, the Couchbase Server do not have the concept of collections. Documents do not have fixed schema, multiple documents with different schema can be stored in same bucket. One or more attributes are added to differentiate the various objects stored in a bucket and create indexes on them.

Each bucket is split into 1024 logical partition called vBuckets. A vBucket is treated as owner of subset of key. Every key belongs to a vBucket.

Data Model A document is the most basic unit of data manipulation in Couchbase Server. All the documents are stored in JSON format without predefined schema.

Querying and Indexing Couchbase Server's views are responsible for querying data from bucket. Views are defined in a specific kind of document called *design document*. These documents are bounded to a single bucket, hence cannot executes from others. The design document holds JavaScript code that implements *Mapreduce* operations which creates view's index in user defined format. The *map()* function in design document filters and extracts information from the document stored in bucket. Each document in Couchbase bucket for a view is submitted to the *map()* function at once. After filtration, the output is a set of key/value pairs is the result of view. The *map*function are supplied with two parameters. First, the JSON document data store in bucket and optional second parameter is metadata associated with the document. The map function's output can be zero or more "rows" according to the filer used in *emit()* function. Each *emit()* function returns a single row but can be called multiple times inside a single map function. The *reduce()* function is used to aggregate the content generated in map phase. Couchbase has built-in reduce functions like *_count*, *_sum* and *_stats*.

Table 2.1: Mapreduce in Couchbase Server

map()	*reduce()*

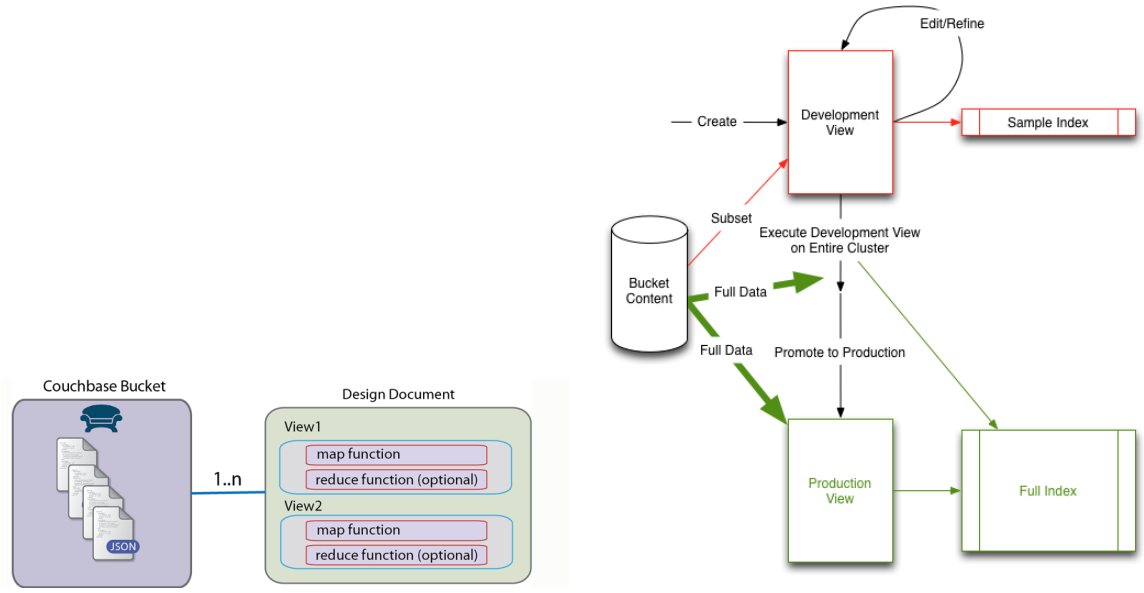
function (doc, meta) {
if(doc.doctype && doc.doctype==="
closed_auctions"){
if(doc.price){
if(doc.price >= 40) {
emit(doc.id,doc.price)
}
}
}
}
	_count

2.2.3 RethinkDB

RethinkDB (Rethinkdb, 2015 (accessed January 12, 2015) is distributed database system to store JSON documents. It has query languages named ReQL which automatically parallelize queries in multiple machines. ReQL is based on three main principle: it is completely embedded with programming language, ReQL queries can be passed as pipeline from one stage to another to get required result that means it is possible to use series of simple queries together to perform complex operation. Finally, all the queries are executed in server without any intermediate network round trip required the server and clients.

Data Model Rethinkdb stores JSON documents with binary on disk serialization. The data types supported by RethinkDB is same as of JSON. There are two ways to model relationship between the documents in RethinkDB.

1. In *Embedded arrays*, the related sub-documents are inserted with specific key



(a) Views in Document Design (b) View's Workflow

Figure 2.4: Couchbase Server's Document Design

in side a document as in MongoDB 2.2.1. The advantages of using embedded arrays are:

- The Queries are tend to be simpler.
- If a dataset is large and don't fit in RAM, data is loaded faster from disk as compare to tables.

Disadvantages of embedded arrays are:

- Everything should be loaded in memory for any operation.

2. In *multiple tables* approach, document with similar schema are stored in tables, like in relational database, connect by reference key to another table. Unlike embedded document, operation of a table don't required to load data from reference table. Table has also some disadvantages like liking between table is more complicated.

Query Model ReQL is the RethinkDB's query language. In ReQL, the JavaScript expression can be embedded at anywhere as a part of query as it embeds into programming language. The lamda function is a part of query or sub-queries that gives more flexibility for expected result. All the ReQL queries are chainable, at any part of query the with {.} operator can be extended at any level as in Code 2.4.

```
r.table("users").run(conn)
r.table("users").pluck("last_name").run(conn)
r.table("users").pluck("last_name").distinct().run(conn)
r.table("users").pluck("last_name").distinct().count().run(conn)
```

Code 2.4: Chainable Query in ReQL

Unlike other NoSQL database, it supports join queries like in relational database in one to many or many to many based on requirement of application in distributed way.

Indexing RethinkDB uses B-Tree to store indexes. At the time of table creation, it provides a option to specify the attribute as a primary key. By default, *id* is behave as primary key if nothing is specified. At the time data insertion, if specified primary key attribute exists, it's value is used to index the document. Otherwise, a random unique string is generated to index automatically. Beside primary index, RethinkDB support compound, secondary, geospatial and arbitrarily computed indexes. Every RethinkDB query, including update operations, uses one and only one index(Rethinkdb, 2015 (accessed January 12, 2015).

- Simple indexes are created to get or sort efficiently by a value of a an attribute.
- Compound indexes, as name suggest, are used to to retrieve document by multiple fields.
- Multi indexes are the indexes for array value of a field.
- Indexes based on arbitrary expression can be used to create index at any type of user defined expressions like lamda functions.

Chapter 3

Semi-structured Data: XML and JSON

XML, by definition a textual markup language where data elements are ordered by nature: *string* is core data type from which richer data types e.g. integers, floats and user-defined abstract data types are derived (Schmidt et al., 2002). A JSON or JavaScript Object Notation is programming language model. It is minimal textual and a subset of JavaScript. JSON document consists of two data structures:

- Objects, an unordered collection of name-value pair that are encapsulated by curly braces { and }. The key is a string encapsulated in double quotes and must followed by a column(:) to it's value. The key must be unique for each object.
- Arrays, which are an ordered list of values

A JSON value can be Object, Array, number, string, `true`, `false` and `null`.

3.1 Problem to translate from XML to JSON

JSON and XML look conceptually similar as they both text based markup language, which are designed to represent data in human-readable form, exchangeable across multiple platforms and parseable by common programming language. When we look at them first, they appear to be quite similar, with difference only in their syntax. But it turns out that they are fundamentally incompatible, as we will see in the following (Lee, 2011).

Root node and anonymous values Each XML document have one and only one root node. JSON support the anonymous values(also refer as string value) that don't need key-value pair. For example 3.1 is valid JSON. In practice, XML root node is implicitly created in the model and will not have a textual representation.

Table 3.1: Anonymous values of JSON in XML

JSON	XML

`["Hello World"]`	`<root>Hello World</root>`
	or
	`<root value="Hello World">`

Arrays Arrays are native data types of JSON which does not exists in XML. There is no direct markup for array of Object from JSON to XML.

Identifiers XML is much more restrictive for identifiers as compare to JSON which allows any string to be an identifier. Translating from XML to JSON do not cause problem but in reverse case might lead to invalid XML. For example "Hello World" is a valid identifier in JSON but not valid attribute or element in XML as there exists a whitespace between two words.

Attributes JSON does not have the concept or representation for XML's attributes. When mapping from XML to JSON, attributes are translated to name object member along with other child elements. This information will be loss when mapping from JSON to XML.

Namespaces XML supports the concept of namespaces to provide uniqueness in element and attributes in XML document. Namespaces do not exists in JSON. Mapping QNames in XML with namespaces to JSON can lead ambiguous and duplicate names.

Others There are also some problem like processing instructions and comments which XML supports but not JSON. Other issues for example character set and encoding are not easily exchangeable.

3.2 Mapping

XML and JSON have different data types. Compare to JSON, XML has more flexible data types. In Table 3.2 is given types of data for simple XML and relevant JSON type.

Table 3.2: Translation of simple XML data types into JSON

XML type definition	JSON type definition
xs:string	{ "type": "string" }
xs:boolean	{ "type": "boolean" }
xs:float xs:double xs:decimal xs:integer (All Other Numbers)	{ "type": "number" }
(Remaining all others)	{ "type": "string" }

For complex XML data type, there are two options in JSON, either object or array based on XML object.

3.3 Migration from XML to JSON

It takes different stages to convert XML data to JSON.

Step1.
XML to JSON friendly XML: Convert standard XML data to JSON friendly XML.

Step2.
Data type of XML to JSON type: Define the data type of XML element whether it is array type or Object type or other scaler type.

Step3.
Final steps: Map XML to JSON object or array.

Step 1
At first, all attributes of XML document are represented as ordered(first) child element of parent then attributes are deleted. There can be more than one attributes which are inserted in order. XML node can contain both text node and element node as child, but JSON can have only key value pair. so each of text node which also has element node as siblings moved to "childtext"(name of the element) node. At the end of first step, if an element has another element as child node, then it is represented as Object type of JSON. It is necessary to identify if a siblings of an element node(S) has same element name or not. If this condition exists, S is represented as the array in JSON. All the child nodes of all siblings are moved inside S node. If S is already object type then it is replace with array type else new array is added.

Algorithm 1: Pseudocode to convert normal XML to JSON friendly XML

```
Initialize D = "XMLdocument";
for all descendant-or-self node of D, X which has attributes A do
  move all attributes A to ordered child element of X
end for
for all descendant-or-self node of D,X do
  if X contains Text node and element node Both then
    create new element "childtext"
    move text node to "childtext" element
  end if
end for
for all descendant node of D,X do
  for all child element C in X do
    if C has siblings S with same name then
      convert C as Array type
      move child of S into C as <_> element
    end if
    if C has child element then
      convert C as Object type
    end if
  end for
end for
```

Algorithm 2: convert data type of XML to JSON data type(Step 2)

```
Initialize D = "XMLdocument";
for all descendant-or-self node of D as C do
  if C has no attribute "type" then
    get content of C
    identify content type according to Table 3.2. and add attribute to C
    "type"
  end if
end for
```

Table 3.3: XML to JSON friendly XML(STEP 1)

XML

```
<info>
  <name>
    <f>a</f>
  </name>
  <age>24</age>
  <ismarried>false</ismarried>
  <city name="Armonk"/>
  <state>NY</state>
  <contact>
        home
    <p>993−330</p>
    <p>993−331</p>
  </contact>
</info>
```

Algorithm 1

```
<info type="object">
  <name type="object">
    <f>a</f>
  </name>
  <age>24</age>
  <ismarried>false</ismarried>
  <city type="object">
    <name>Armonk</name>
  </city>
  <state>NY</state>
  <contact type="object">
        <childtext>home</childtext>
    <p type="array">
        <_>993330</_>
        <_>993−331</_>
    </p>
  </contact>
</info>
```

Table 3.4: XML to JSON friendly XML(STEP 2)

Algorithm 1

```
<info type="object">
  <name type="object">
    <f>a</f>
  </name>
  <age>24</age>
  <ismarried>false</ismarried>
  <city type="object">
    <name>Armonk</name>
  </city>
  <state>NY</state>
  <contact type="object">
        <childtext>home</childtext
          >
    <p type="array">
        <_>993330</_>
        <_>993−331</_>
    </p>
  </contact>
</info>
```

Algorithm 2

```
<info type="object">
  <name type="object">
    <f type="string">a</f>
  </name>
  <age type="number">24</age>
  <ismarried type="boolean">false</ismarried>
  <city type="object">
    <name type="string">Armonk</name>
  </city>
  <state>NY</state>
  <contact type="object">
        <childtext type="string">home</childtext>
    <p type="array">
        <_ type="number">993330</_>
        <_ type="string">993−331</_>
    </p>
  </contact>
</info>
```

Step 3 After Step 1. and 2., a complete JSON friendly XML is generated. All XML object and array are mapped to <*key*, *value*> pair of JSON and their respec-

```
<info type="object">
  <name type="object">
    <f type="string">a</f>
  </name>
  <age type="number">24</age>
  <ismarried type="boolean">false</ismarried>
  <city type="object">
    <name type="string">Armonk</name>
  </city>
  <state type="string">NY</state>
  <contact type="object">
        <childtext type="string">home</childtext>
    <p type="array" >
          <_ type="number">993330</_>
          <_ type="string">993-331</_>
    </p>
  </contact>
</info>
```

```
{
    "info":{
      "name":{
        "f":"a"
      },
      "age":24,
      "ismarried":false,
      "city":{
        "name":"Armonk"
      },
      "state":"NY",
      "contact":{
          "childtext":"home",
        "p":[
          993330,
          "993-331"
        ]
      }
    }
}
```

3.4 XMark

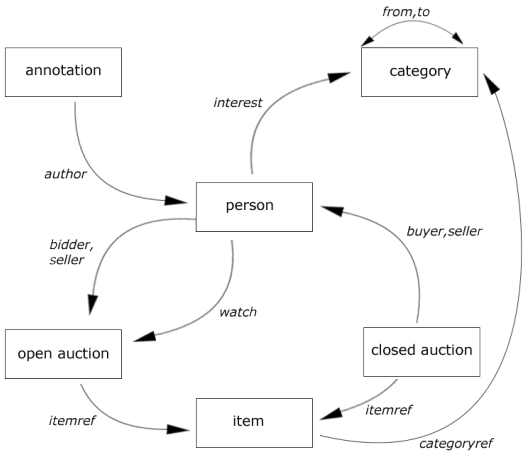
The XML benchmarking project XMARK (Schmidt et al., 2002) is one of the most popular and most commonly used XML benchmarking project to date. It provides a small executable tool called *xmlgen* that allows for the creation of a synthetic XML dataset based on a fixed schema describing an Internet auctions database. Its generator can be used to build a single record with a large, hierarchic XML tree structure. A factor can be specified to scale the generated data, ranging from a few kilobytes to any arbitrary size limited by the capacity of the system. The textual part of resulting XML document is constructed from 17,000 most frequently occurring words of Shakespeare's plays.

3.4.1 Dataset

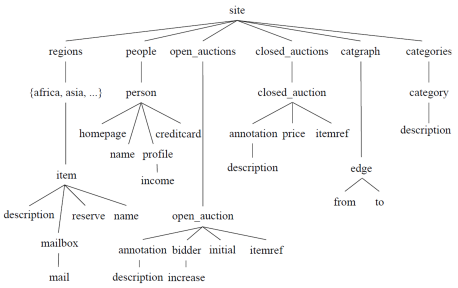
The main entities of XMark data come in two groups. *person*, *open_auction*, *closed_auction*, *item* and *category* are categorized as first group. The second group entities *annotation* and *description* are natural language text and document-centric element structure.The relationship between the entities in group one are expressed in reference and second group entities are embedded into subtree of group one's entities. As shown in Fig. 3.1b, Xmark dataset has following properties:

- *people* are the collection of *person* that are connected to buyer and seller of open_auctions, closed_auctions etc. Each person has an unique *id* to reference to another entities like open and closed auctions.

- *items* are the objects for sale or already sold. Each *item* carries a unique identifier and has properties like name, payment information (credit card, money order), description, a reference to the seller etc., all are encoded as



(a) Reference in *XMark*



(b) Reference in *XMark* dataset tree (Schmidt et al., 2002)

Figure 3.1: XMark data tree and reference

elements. Each item belongs to the world's regions `africa`, `asia`, `australia`, `europe`, `namerica` and `samerica` as a parent of an item element.

- *open_auctions* refer to currently in progress auctions which contain bid history(increase/decrease over time) with reference to bidder and seller reference, current bid, the time interval within which the bid is accepted, the status of transaction, a reference to the item which is being sold etc.

- *closed_auctions* contains auctions that are successfully completed bidding. They have properties like buyer and seller information, reference to *person*, a reference to an item that is sold, amount of price, quantity of items sold, date of transaction, type of transaction.

- *categories* are used to classify the items. Each category has an unique identifier which is used to reference in item, a name and a description.

- A `catgraph` links categories into a network. The full semantics of XMark dataset can be found in Schmidt et al. (2002).

d to create

3.4.2 Queries

e style of
d complete the

The XMark project contains 20 XQuery Queries which focus on various aspect of language such as aggression, reference, ordering, wildcard expressions, joins, user defined functions etc.(Mlỳnková, 2008). In this section these queries has been reprinted. .....

3.4.3 XMark data into NoSQL Database

A synthetic XMARK dataset consists of one(huge) record in tree structure (Wang et al., 2003). However, as already mentioned in 3.4, each subtree in schema, *items*, *people*, *open_auctions* , *closed_auctions*,*catgraph* and *categories* contains large number of instances in the database which are indexed. In most of NoSQL system,

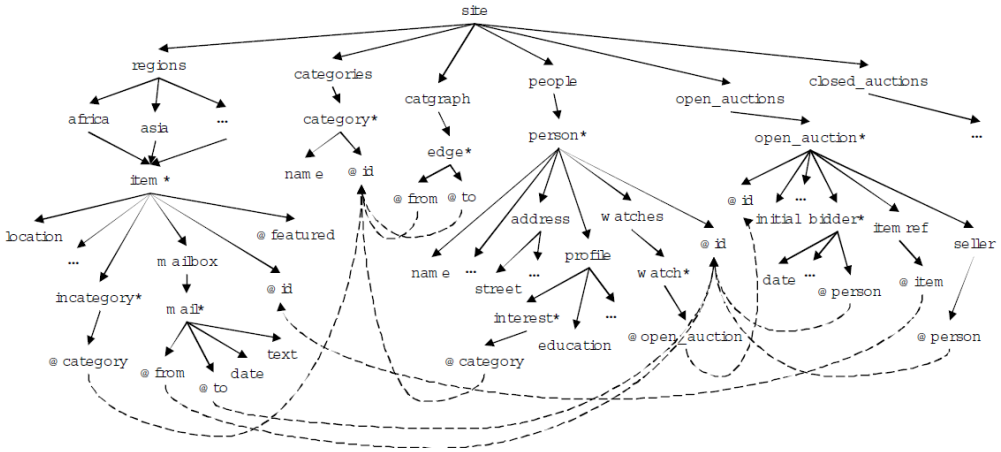


Figure 3.2: XMark ER-Diagram. Nodes, solid arrows, and dashed arrows represent schema elements (or attributes, with prefix '@'), structural links, and value links, respectively. Elements with suffix '*' are of SetOf type(Yu and Jagadish, 2006)

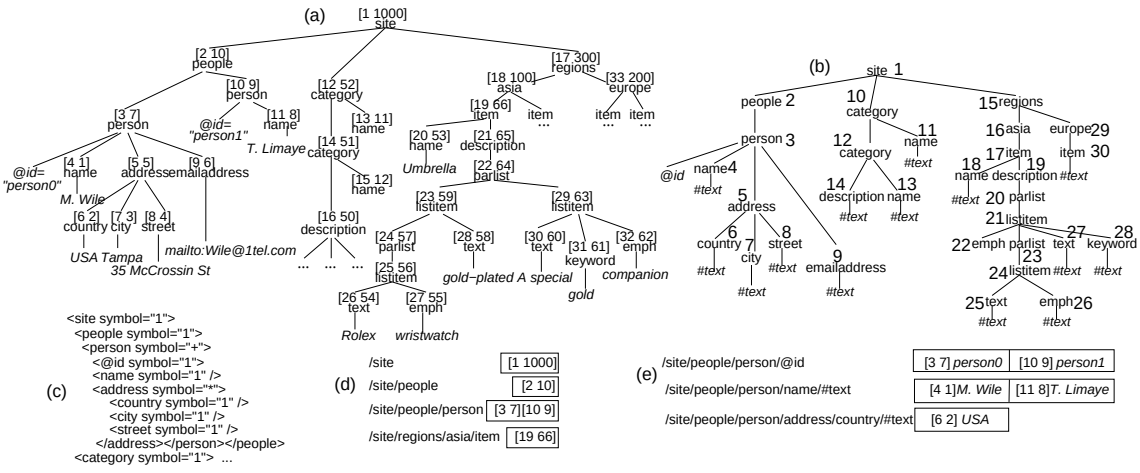


Figure 3.3: XMark document snippet, its path summary, and some path-partitioned storage structures.

this scenario is different, each instance has it's own index structure, the dataset cannot be in just a huge block. As the data model of NoSQL do not match this single structure-encoded sequence, we breakdown it's tree structure into set of sub-structure without losing the overall data and create index for each of them. Beside this, each NoSQL database has their own data model design, we need to define model design for each of those databases separately.

The generalized concept of XMark data into NoSQL database is given here, but it might be slightly different from each of them. All sub-structures or collections *items*, *people*, *open_auctions*, *closed_auctions*, *catgraph* and *categories* are the basic for the document fragmentation. Each of these collection stores respective first group entities *item*, *person*, *open_auction*, *closed_auction* and *category* as mentioned in 3.4. These entities represent the documents of NoSQL database. In each of these documents, one special field **type** is added to represent the collection. For example, in case of person collection *type* will be *people*. This key value set will be also the part of document as given in . There is one exceptional case for *items* which has **regions** as grandparent and name of different regions like *asia*, *europe* etc. as parent. The *type* for *item* document will be *"items"* as others, one extra field need to be added to represent each region, so there will be one more field *regions* as in Table 3.6.

Table 3.6: Extra added attribute in document

for *person*	for items which has region name *asia*
`{ "type":"people" }`	`{ "type":"items", "regions":"asia" }`

Table 3.7: Example: XMARK data in XML and JSON format

person0 in XML

```
<people>
  <person id="person0">
    <name>Kasidit Treweek</name>
    <emailaddress>mailto:Treweek@cohera.
        com</emailaddress>
    <phone>+0 (645) 43954155</phone>
    <homepage>http://www.cohera.com/~
        Treweek</homepage>
    <creditcard>9941 9701 2489 4716</
    creditcard>
    <profile income="20186.59">
      <interest category="category251" />
      <education>Graduate School</
          education>
      <business>No</business>
    </profile>
  </person>
</people>
```

person0 in JSON for a NoSQL database

```
{
    "id": "person0",
    "type": "people",
    "name": "Kasidit Treweek",
    "emailaddress": "mailto:Treweek@cohera.com",
    "phone": "+0 (645) 43954155",
    "homepage": "http://www.cohera.com/~Treweek",
    "creditcard": "9941 970124894716",
    "profile": {
        "income": 20186.59,
        "interest": [{
            "category": "category251"
        }],
        "education": "Graduate School",
        "business": "No"
    }
}
```

3.4.3.1 XMark data in JSON for NoSQL

3.4.3.2 XMARK in MongoDB

In section 2.2.1, it was explained the relationship between collection and documents. MongoDB's collections have similar and related documents together. As we habe seen in Fig. 3.2, we create each collection of each substructure in such a way that we will not lose data as well as most representation of xmark data. For Mongodb, the

data model of in 3.4.3 slightly change. Each *type* represented as a collection. So we will have six collections and type is already represented by the collections. we don't need key/value of type in our document. For *items*, *regions* contains the name of region of each item as usual.

Finally as we mentioned in Indexing, the *id* attribute of each these documents will be renamed to *_id* for default indexing. In case of `closed_auctions` and `catgraph`, system will automatically generate `_id` which is not applicable for us right now.

```
{
        "_id": "person0",
        "name": "Kasidit Treweek",
        "emailaddress": "mailto:Treweek@cohera.com",
        "phone": "+0 (645) 43954155",
        "homepage": "http://www.cohera.com/~Treweek",
        "creditcard": "9941 970124894716",
        "profile": {
                "income": 20186.59,
                "interest": [{
                        "category": "category251"
                }],
                "education": "Graduate School",
                "business": "No"
        }
}
```

Code 3.22: Mongodb data representation of XMARk data

3.4.3.3 Couchbase Server

Document design For Couchbase Server, there is no need to change any structure of XMark NoSQL representation as mentioned in 3.4.3 as there is no concept of fragmentation like in *collections* of Mongodb or *tables* in Rethinkdb. The documents are identified by *type*. All of these documents are inserted in a single Bucket with *id* as key. For those documents that doesn't have id field, will be manually generated.

3.4.3.4 Rethinkdb

RethinkDB Rethinkdb (2015 (accessed January 12, 2015) is distributed database system to store JSON documents that uses efficient query languages named ReQL which automatically parallelize queries in multiple machines. ReQL is based on three main principle: it is completely embedded with programming language, ReQL queries can be passed as pipeline from one stage to another to get required result that means it is possible to use series of simple queries together to perform complex operation. Finally, all the queries are executed in server without any intermediate network round trip required the server and clients.

Data Model The relationships between documents can be can be created by embedded arrays of documents and linking document stores in multiple tables similar to MongoDB in 3.4.3.2. Rethinkdb stores JSON documents with binary on disk serialization. RethinkDB implicitly support JSON data types: `object`, `array`, `number`, `boolean` and `null`. RethinkDB allows embedded JavaScript expressions anywhere as part of query language.

Indexing At the time of table creation, RethinkDB provide option of specifying the attributes that will serve as primary key, by default `id` will serve if primary key attribute is not specified.

Chapter 4

Performance

4.1 Evaluation of Test devices

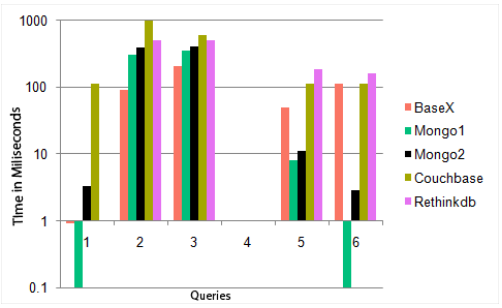
4.2 Results and Analysis

4.2.1 Summary

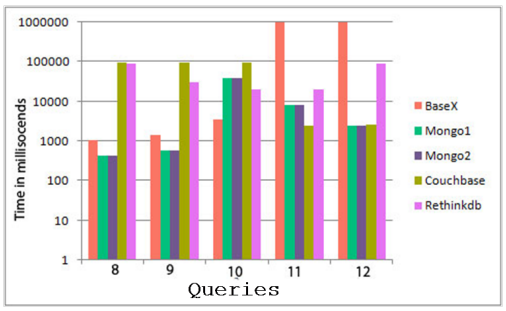
4.3 Discussion

4.4 Conclusion

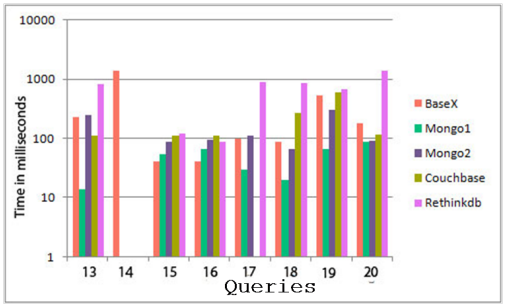
4.5 Future Work



(a) XMark Queries No. 1 to 7



(b) Join Queries (No. 8 to 12)



(c) Queries 13 to 20

Figure 4.1: XMark Queries time in XML and NoSQL databases(Mongo1: Mongodb shell, mongo2:through application)

Appendices

Appendix A

XMARK Queries

A.1 MongoDB

Q1.

```
db.people.find({_id:"person0"},{_id:0,"name":1});
```

Q2.

```
db.open_auctions.aggregate([
        {$project:{_id:1,bidder:"$bidder"}},
        {$unwind: "$bidder"},
        {$group:{_id:"$_id",increase:{$first:"$bidder.
           increase"}}},
        {$sort:{_id:1}},
        {$unwind: "$increase"},
        {$project:{_id:0,increase:1}}
])
```

Q3.

```
    db.open_auctions.aggregate([
        {$unwind: "$bidder"},
        {$group:{_id:"$_id",first:{$first:"$bidder.
           increase"},last:{$last:"$bidder.increase"}}},
        {$unwind:"$first"},
        {$unwind:"$last"},
        {$project:{_id:1,first:1,last:1,diff:{$gte:["$last
           ",{$multiply:["$first",2]}]}}},
        {$match:{diff:true}},
        {$sort:{_id:1}},
        {$project:{_id:0,increase:{first:"$first",last:"
           $last"}}}
    ])
```

Q5.

```
    db.closed_auctions.find({price:{$gte:40}}).count()
```

Q6.

```
    db.regions.find().count()
```

Q8.

```
function q8() {
    var ids = new Array();
    var name = new Array();
    var closed = new Array();
    var i=0;
    db.people.find({},{name:1}).forEach(function(
       people){
        ids[i] = people._id;
        name[ids[i]] = people.name;
        //closed[ids[i]] = {name:people.name,count
           :0};
        i++;
    });
    var closed_auc = db.closed_auctions.aggregate([
        {$project:{_id:1,person:"$buyer.person"}},
        //{$match:{person:{$in:["person12124","
           person12883"]}}},
        {$match:{person:{$in:ids}}},
        {$group:{_id:"$person",count:{$sum:1}}},
        {$sort:{count:-1}},
        {$project:{person:"$_id",count:1,_id:0}}
    ]);
    if(closed_auc) {
        var i=0
        while(closed_auc.hasNext()){
            var auc = closed_auc.next();
            var id = auc.person;
            closed[i] = {"name":name[id],"count
               ":auc.count}
            name[id] = 0;
            i++;
            //printjson(name.indexOf(id));
        }

    }/**/
    var len = closed.length ;
    var j = 0;
    for(var i=0; i < ids.length; i++) {
        if(name[ids[i]] !== 0) {
        //printjson(i);
        closed[len+j] = {"name":name[ids[i]],"
           count":0};
        j++;
        }
    }
    //printjson("Execution Time" + end-start);
    return closed;

}
```

Q10.

```
function q10() {
    var debugId = "person3"
    var ids = new Array();
    var i = 0;
    var allcategories = new Array();
    db.people.aggregate([
    {$match:{"profile.interest":{$exists:true}}},
    {$project:{_id:1, interest:"$profile.interest"}},
    {$unwind:"$interest"},
    {$group:{_id:"$interest.category"}},
    {$project:{_id:0, category:"$_id"}}
    ]).forEach(function(people){
            var catId = people.category;
            ids[i] = {categorie:{id:catId,profile:
               getProfileByCategory(catId)}}
            i++;
    });
    //printjson("Execution Time" + diff);
    return ids;
}
```

Q11.

```
function q11() {
var start = new Date().getTime();
var debugId = "person3"
var ids = new Array();
var open_auc = function(initial){
    return db.open_auctions.find({initial:{$lt:initial
       }},{_id:1}).count();
}

var i=0;
db.people.find({},{_id:1, name:1,"profile.income":1}).
   forEach(function(people){
    var income = ((people.profile) && people.profile.
       income)? people.profile.income/5000:0;
    ids[i] = {item:{name: people.name,id:people._id,
       count:open_auc(income)}};
    i++;
});
return ids;
}
```

Q12.

```
function q12() {
var debugId = "person3"
var ids = new Array();
var open_auc = function(initial){
    return db.open_auctions.find({initial:{$lt:initial
       }},{_id:1}).count();
}
var i=0;
```

```
db.people.find({"profile.income":{$exists:true}, "profile
    .income":{$gt:50000}},{_id:1, name:1,"profile.income":
    1}).forEach(function(people){
        var income = ((people.profile) && people.profile.
            income)? people.profile.income/5000:0;
        ids[i] = {item:{name: people.name,id:people._id,
            count:open_auc(income)}};
        i++;
});
return ids;

}
```

Q13.

```
db.regions.find(
    {regions:"australia"},
    {_id:0,name:1,description:1}
)
```

Q14.

```
db.regions.aggregate([
    {$match:{$text:{$search:"gold"}}},
    {$group:{_id:"$_id"}},
    {$group:{_id:null,count:{$sum:1}}},
    {$project:{_id:0,count:1}}
])
```

Q15.

```
db.closed_auctions.aggregate([
    {$project:{_id:1,parlist:"$annotation.description.
        parlist"}},
    {$unwind: "$parlist"},
    {$unwind: "$parlist.listitem.parlist"},
    {$project:{_id:1,text:"$parlist.listitem.parlist.
        listitem.text"}},
    {$match:{$or:[{"text.emph.keyword":{$exists:true}}
        ,{"text.emph.keyword.childtext":{$exists:true}
        },{"text.emph.keyword.child":{$exists:true}}]}
        },
    {$project:{_id:0,text:"$text.emph.keyword"}}
])
```

Q16.

```
db.closed_auctions.aggregate([
    {$project:{_id:1,parlist:"$annotation.description.
        parlist",person:"$seller.person"}},
    {$unwind: "$parlist"},
    {$unwind: "$parlist.listitem.parlist"},
    {$project:{_id:1,text:"$parlist.listitem.parlist.
        listitem.text",person:1}},
    {$match:{"text.emph.keyword":{$exists:true}}},
    {$project:{_id:0,"person.id":"$person"}}
])
```

Q17.
```
db.people.find({homepage:{$exists:false}},{_id:0,person
    :"$name"})
])
```

Q18.
```
db.open_auctions.aggregate([{$match:{reserve:{$exists:
    true}}},
    {$project:{_id:0,reserve:{$multiply:["$reserve",2.
        20371]}}}])
```

Q19.
```
db.regions.aggregate([{$sort:{location:1}},
    {$project:{_id:0,"item.name":"$name","item.
        location":"$location"}}])

db.people.aggregate([
    {$project:{_id:1,p:"$profile.income"}},
    {$project:{_id:1,con:
            {$cond:[
              {$gt:["$p",100000]},"preferred",
              {$cond:[{ $and: [{$lt:["$p", 100000]},{
                  $gte:["$p", 30000]}]},"standard",
    {$cond:[{$and:[{$lt:["$p",30000]},{$gt:["$p",0]}]}
        ,"challange","na"]}
    ]}
    ]}
    }},
    {$group:{_id:"$con",sum:{$sum:1}}},
    {$project:{_id:1,sum:1}}])
```

A.2 RethinkDB

Q1.
```
r.table("people").get("person0")("name")
```

Q2.
```
r.table("open_auctions").map(function(doc) {return {
    increase:doc("bidder").nth(0)("increase").nth(0).
    default("")}})
```

Q3.
```
r.table("open_auctions").hasFields("bidder").filter(
            r.row("bidder").nth(0)("increase").nth(0).
                mul(2).le(r.row("bidder").nth(r.row("
                bidder").count().sub(1))("increase").
                nth(0))
        ).map(function(doc){
          var last = doc("bidder").count().sub(1);
          return {increase:{first:doc("bidder").nth(
              0)("increase").nth(0), last:doc("bidder
              ").nth(last)("increase").nth(0)} }
        })
```

Q5.
```
r.table("closed_auctions").count(function(f){
            return f("price").default(0).gt(40)
          })
```

Q6.
```
r.table("regions").count()
```

Q8.
```
r.table("people").map(function(d){
            var x = r.table("closed_auctions").getAll(
               d("id"),{index:"buyer_person"}).count()
               ;
            return {person:d("name"),value:x}
          })
```

Q9.
```
var d1= "0001";
r.db(d1).table("people").outerJoin(r.db(d1).table("
   closed_auctions"),function(p,c){
 return p("id").eq(c("buyer")("person"))
}).map({person:r.row("left")("id"),name:r.row("left")
   ("name"),item:r.row("right")("itemref")("item").
   default("")})
  .outerJoin(r.db(d1).table("regions").getAll("europe"
     ,{index:"regions"}),function(pc,r){
  return pc("item").eq(r("id"))
  }).map(function(d){


    //return (d("right") && d("right")("id")) ? {
       person:d("left")("person"),name:d("left")("name
       "),item:d("right")("id")} :{person:d("left")("
       person"),name:d("left")("name")}


     return {person:d("left")("person"),name:d("left")
        ("name"),item:d("right")("id").default("0")}
  }).group('person','name').ungroup().map(function(d){

    return {name: d('group')(1), total: d('reduction')
       .count(function(item){
      return item("item")
    })
    }
  }).orderBy(r.desc('total'))
```

Q11.
```
r.table("people").pluck("id",{profile:{income:true}})
            .map(function(f){
              var i = f("profile")("income").default(0)
                 .div(5000);
              return {
```

```
                    id:f("id"),
                    item:r.table("open_auctions").filter(
                        function(m){
                    return m("initial").default(0).lt(i)
                        }).count()
                    }
                })
```

Q12.

```
r.table("people").filter(function(f){return f("profile
    ")("income").default(0).gt(50000)})
            .map(function(f){
            return {
              id:f("id"),
              item:r.table("open_auctions").count(
                  function(x){
                return f("profile")("income").default(
                    0).gt(x("initial").default(0).mul(5
                    000))
              })
            }
          })
```

Q13.

```
r.table("regions").getAll("australia",{index:"regions"
    }).map({item:{name:r.row("name"),description:r.row
    ("description")}})
```

Q16.

```
r.table("closed_auctions").hasFields({annotation:{
    description:{parlist:true}}})
        ("annotation")("description").concatMap(r.
            row("parlist")).hasFields({listitem:{
            parlist:true}})("listitem").concatMap(r.
            row("parlist")).hasFields({listitem:{text
            :{emph:{keyword:true}}}})
```

Q17.

```
r.table("people").filter(function(d){return d.hasFields
    ("homepage").not()}).map({person:{name:r.row("name")
    }})
```

Q18.

```
r.table("open_auctions").hasFields("reserve").map(r.row
    ("reserve").mul(2.20371))
```

Q19.

```
r.table("regions").orderBy({index:"location"}).pluck("
    location","name")

r.table("people").map(function(doc){return {income:doc
    ("profile")("income").default(0)}})
      .map({
```

```
 preferred:r.branch(r.row("income").ge(100000)
     ,'preferred',0),
  standard:r.branch(r.row("income").le(100000).
     and(
    r.row("income").ge(30000)
  ),'standard',0),
  challenge:r.branch(r.row("income").le(30000).
     and(
    r.row("income").gt(0)
  ),'challenge',0),
  na:r.branch(r.row("income").le(0),'na',0)
}).group('preferred','standard','challenge','na
    ').count()
```

A.3 Couchbase

Bibliography

Q20Rick Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4): 12–27, 2011.

Xiaoming Gao. Investigation and comparison of distributed nosql database systems.

Christian Grün. BaseX – The XML Database for Processing, Querying and Visualizing large XML data. http://basex.org, October 2010.

Christoforos Hadjigeorgiou et al. RDBMS vs NoSQL: Performance and scaling comparison. *The University of Edinburgh*, 2013.

Robin Hecht and S Jablonski. Nosql evaluation. In *International Conference on Cloud and Service Computing*, 2011.

Haifeng Jiang, Hongjun Lu, Wei Wang, and Jeffrey Xu Yu. XParent: An efficient RDBMS-based XML database system. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 335–336. IEEE, 2002.

David Lee. Jxon: an architecture for schema and annotation driven json/xml bidirectional transformations. In *Proceedings of Balisage: The Markup Conference*, 2011.

Kyle Lichtenberg and Miles Ward. Nosql database in the cloud: Couchbase server 2.0 on aws. 2013.

Irena Mlỳnková. Xml benchmarking: Limitations and opportunities. Technical report, Technical Report, Department of Software Engineering, Charles University, Czech Republic, 2008.

Cory Nance, Travis Losser, Reenu Iype, and Gary Harmon. NoSQL vs RDBMS-why there is room for both. In *Proceedings of the Southern Association for Information Systems Conference*, pages 111–116, 2013.

Kai Orend. Analysis and classification of nosql databases and evaluation of their ability to replace an object-relational persistence layer. *Architecture. Citeseer*, page 100, 2010.

David Ostrovsky and Yaniv Rodenski. *Pro Couchbase Server*. Apress, 2014.

Rethinkdb. *An open-source distributed database built with love - RethinkDB*, 2015 (accessed January 12, 2015). URL `http://www.rethinkdb.com/`.

Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J Carey, Ioana Manolescu, and Ralph Busse. Xmark: A benchmark for xml data management. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 974–985. VLDB Endowment, 2002.

Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J DeWitt, and Jeffrey F Naughton. Relational databases for querying XMl documents: Limitations and opportunities. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 302–314. Morgan Kaufmann Publishers Inc., 1999.

Haixun Wang, Sanghyun Park, Wei Fan, and Philip S. Yu. Vist: A dynamic index method for querying xml data by tree structures. In *In SIGMOD*, pages 110–121, 2003.

Cong Yu and HV Jagadish. Schema summarization. In *Proceedings of the 32nd international conference on Very large data bases*, pages 319–330. VLDB Endowment, 2006.

List of Figures

List of Tables

Listings