CHAPTER 1: INTRODUCTION

1.1 Relational Databases History

A relational database is a collection of data items organized as a set of formally described tables from which data can be accessed easily. A relational database is created using the relational model. The software used in a relational database is called a relational database management system (RDBMS). A relational database is the predominant choice in storing data, over other models like the hierarchical database model or the network model.

1.2 Relations or Tables

A relation is defined as a set of tuples that have the same attributes. A tuple usually represents an object and information about that object. Objects are typically physical objects or concepts. A relation is usually described as a table, which is organized into rows and columns. All the data referenced by an attribute are in the same domain and conform to the same constraints. The relational model specifies that the tuples of a relation have no specific order and that the tuples, in turn, impose no order on the attributes. Applications access data by specifying queries, which use operations such as select to identify tuples, project to identify attributes, and join to combine relations. Relations can be modified using the insert, delete, and update operators. New tuples can supply explicit values or be derived from a query. Similarly, queries identify tuples for updating or deleting. It is necessary for each tuple of a relation to be uniquely identifiable by some combination (one or more) of its attribute values. This combination is referred to as the primary key.

1.3 Base and derived relations

In a relational database, all data are stored and accessed via relations. Relations that store data are called "base relations", and in implementations are called "tables". Other relations do not store data, but are computed by applying relational operations to other relations. These relations are sometimes called "derived relations". In implementations these are called "views" or "queries". Derived relations are convenient in that they act as a single relation, even though they may grab information from several relations. Also, derived relations can be used as an abstraction layer.

1.3.1 Domain

A domain describes the set of possible values for a given attribute, and can be considered a constraint on the value of the attribute. Mathematically, attaching a domain to an attribute means that any value for the attribute must be an element of the specified set. The character data value 'ABC', for instance, is not in the integer domain. The integer value 123, satisfies the domain constraint.

1.4 Constraints

Constraints make it possible to further restrict the domain of an attribute. For instance, a constraint can restrict a given integer attribute to values between 1 and 10. Constraints provide one method of implementing business rules in the database. SQL implements constraint functionality in the form of check constraints. Constraints restrict the data that can be stored in

relations. These are usually defined using expressions that result in a boolean value, indicating whether or not the data satisfies the constraint. Constraints can apply to single attributes, to a tuple (restricting combinations of attributes) or to an entire relation. Since every attribute has an associated domain, there are constraints (domain constraints). The two principal rules for the relational model are known as entity integrity and referential integrity. (**"**Referential integrity is the state in which all values of all foreign keys are valid. Referential integrity is based on entity integrity. Entity integrity requires that each entity have a unique key. For example, if every row in a table represents relationships for a unique entity, the table should have one column or a set of columns that provides a unique identifier for the rows of the table. This column (or set of columns) is called the parent key of the table. To ensure that the parent key does not contain duplicate values, a unique index must be defined on the column or columns that constitute the parent key. Defining the parent key is called entity integrity**"**)

1.5 Primary Keys

A primary key uniquely defines a relationship within a database. In order for an attribute to be a good primary key it must not repeat. While natural attributes are sometimes good primary keys, surrogate keys are often used instead. A surrogate key is an artificial attribute assigned to an object which uniquely identifies it (for instance, in a table of information about students at a school they might all be assigned a student ID in order to differentiate them).

The surrogate key has no intrinsic (inherent) meaning, but rather is useful through its ability to uniquely identify a tuple. Another common occurrence, especially in regards to N:M cardinality is the composite key. A composite key is a key made up of two or more attributes within a table that (together) uniquely identify a record. (For example, in a database relating students, teachers, and classes. Classes could be uniquely identified by a composite key of their room number and time slot, since no other class could have exactly the same combination of attributes. In fact, use of a composite key such as this can be a form of data verification, albeit a weak one.)

1.6 Foreign Key

A foreign key is a field in a relational table that matches the primary key column of another table. The foreign key can be used to cross-reference tables. Foreign keys need not have unique values in the referencing relation. Foreign keys effectively use the values of attributes in the referenced relation to restrict the domain of one or more attributes in the referencing relation. A foreign key could be described formally as: "For all tuples in the referencing relation projected over the referencing attributes, there must exist a tuple in the referenced relation projected over those same attributes such that the values in each of the referencing attributes match the corresponding values in the referenced attributes."

1.7 Key/Value (or NoSQL) Stores

In computing, NoSQL (mostly interpreted as "not only SQL") is a broad class of database management systems identified by its non-adherence to the widely used relational database management system model, that is NoSQL databases are not primarily built on tables, and as a result, generally do not use SQL for data manipulation.

The following characteristics are often associated with a NoSQL database:

It does not use SQL as its query language

NoSQL database systems rose alongside major internet companies, such as Google, Amazon, and Facebook, which had significantly different challenges in dealing with huge quantities of data that the traditional RDBMS solutions could not cope with (although most of Facebook's infrastructure is based on MySQL databases and so is Twitter's). NoSQL database systems are developed to manage large volumes of data that do not necessarily follow a fixed schema. Data is partitioned among different machines (for performance reasons and size limitations) so JOIN operations are not usable.

It may not give full ACID guarantees

Usually only eventual consistency is guaranteed or transactions limited to single data items. This means that given a sufficiently long period of time over which no changes are sent, all updates can be expected to propagate eventually through the system.

It has a distributed, fault-tolerant architecture

Several NoSQL systems employ a distributed architecture, with the data held in a redundant manner on several servers. In this way, the system can easily scale out by adding more servers, and failure of a server can be tolerated. This type of database typically scales horizontally and is used for managing big amounts of data, when the performance and real-time nature is more important than consistency (as indexing a large number of documents, serving pages on high-traffic websites, and delivering streaming media).

NoSQL database systems are often highly optimized for retrieve and append operations and often offer little functionality beyond record storage (e.g. key-value stores). The reduced run time flexibility compared to full SQL systems is compensated by significant gains in scalability and performance for certain data models.

Key-value, hierarchical, map-reduce, or graph database systems are much closer to implementation strategies, they are heavily tied to the physical representation. The primary reason to choose one of these is if there is a compelling performance argument and it fits your data processing strategy very closely. Beware, ad-hoc queries are usually not practical for these systems, and you're better off deciding on your queries ahead of time.

Relational database systems try to separate the logical, business-oriented model from the underlying physical representation and processing strategies. This separation is imperfect, but still quite good. Relational systems are great for handling facts and extracting reliable information from collections of facts. Relational systems are also great at ad-hoc queries, which the other systems are notoriously bad at. That's a great fit in the business world and many other places. That's why relational systems are so prevalent.

If it's a business application, a relational system is almost always the answer. For other systems, it's probably the answer. If you have more of a data processing problem, like some pipeline of things that need to happen and you have massive amounts of data, and you know all of your queries up front, another system may be right for you.

1.8 Redis

Redis is an open-source, networked, in-memory, key-value data store with optional durability. It is written in ANSI C. The development of Redis is sponsored by VMware.

Languages for which Redis bindings exist include:
ActionScript, C, C++, C#, Clojure, Common Lisp, Dart, Erlang, Go, Haskell, Haxe, Io, Java, server-side JavaScript (Node.js), Lua, Objective-C, Perl, PHP, Pure Data, Python, R, Ruby, Scala, Smalltalk and Tcl.

Data model

In its outer layer, the Redis data model is a dictionary where keys are mapped to values. One of the main differences between Redis and other structured storage systems is that values are not limited to strings. In addition to strings, the following abstract data types are supported:

- Lists of strings
- Sets of strings (collections of non-repeating unsorted elements)
- Sorted sets of strings (collections of non-repeating elements ordered by a floating-point number called score)
- Hashes where keys and values are strings

The type of a value determines what operations (called commands) are available for the value itself. Redis supports high level atomic server side operations like intersection, union, and difference between sets and sorting of lists, sets and sorted sets.

Persistence

Redis typically holds the whole dataset in memory. It is now deprecated, but versions up to 2.4 could be configured to use what they refer to as virtual memory in which some of the dataset is stored on disk. This should not be confused with virtual memory. Persistence is reached in two different ways: One is called snapshotting, and is a semi-persistent durability mode where the dataset is asynchronously transferred from memory to disk from time to time, written in RDB dump format. Since version 1.1 the safer alternative is AOF, an append-only file (a journal) that is written as operations modifying the dataset in memory are processed. Redis is able to rewrite the append-only file in the background in order to avoid an indefinite growth of the journal.

Replication

Redis supports master-slave replication. Data from any Redis server can replicate to any number of slaves. A slave may be a master to another slave. This allows Redis to implement a single-rooted replication tree. Redis slaves are writable, permitting intentional and unintentional inconsistency between instances. The Publish/Subscribe feature is fully implemented, so a client of a slave may SUBSCRIBE to a channel and receive a full feed of messages PUBLISHed to the master, anywhere up the replication tree. Replication is useful for read (but not write) scalability or data redundancy.

Performance

When the durability of data is not needed, the in-memory nature of Redis allows it to perform extremely well compared to database systems that write every change to disk before considering a transaction committed. There is no notable speed difference between write and read operations.

1.9 Python

Python is a general-purpose, high-level programming language whose design philosophy emphasizes code readability. Python's syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C, and the language provides constructs intended to enable clear programs on both a small and large scale.

Python supports multiple programming paradigms, including object-oriented, imperative and functional programming styles. It features a dynamic type system and automatic memory management and has a large and comprehensive standard library.
Like other dynamic languages, Python is often used as a scripting language, but is also used in a wide range of non-scripting contexts. Using third-party tools, Python code can be packaged into standalone executable programs. Python interpreters are available for many operating systems.

CPython, the reference implementation of Python, is free and open source software and has a community-based development model, as do nearly all of its alternative implementations. CPython is managed by the non-profit Python Software Foundation.

Python was conceived in the late 1980s and its implementation was started in December 1989 by Guido van Rossum at CWI in the Netherlands as a successor to the ABC language (itself inspired by SETL) capable of exception handling and interfacing with the Amoeba operating system. Van Rossum is Python's principal author, and his continuing central role in deciding the direction of Python is reflected in the title given to him by the Python community, Benevolent Dictator for Life (BDFL).
Python 2.0 was released on 16 October 2000, with many major new features including a full garbage collector and support for Unicode. With this release the development process was changed and became more transparent and community-backed.

Python 3.0 (also called Python 3000 or py3k), a major, backwards-incompatible release, was released on 3 December 2008 after a long period of testing. Many of its major features have been backported to the backwards-compatible Python 2.6 and 2.7.

CHAPTER 2: SIMULATING ROWS

2.1 Tables in Relational Database:

In relational databases and flat file databases, a table is a set of data elements (values) that is organized using a model of vertical columns (which are identified by their name) and horizontal rows, the cell being the unit where a row and column intersect. A table has a specified number of columns, but can have any number of rows[citation needed]. Each row is identified by the values appearing in a particular column subset which has been identified as a unique key index.

Table is another term for relations; although there is the difference in that a table is usually a multiset (bag) of rows whereas a relation is a set and does not allow duplicates. Besides the actual data rows, tables generally have associated with them some metadata, such as constraints on the table or on the values within particular columns.

The data in a table does not have to be physically stored in the database. Views are also relational tables, but their data are calculated at query time. Another example are nicknames, which represent a pointer to a table in another database.

2.2 DDL/Data Definition Language

A data definition language or data description language (DDL) is a syntax similar to a computer programming language for defining data structures, especially database schemas.

Many data description languages use a declarative syntax to define fields and data types. SQL, however, uses a collection of imperative verbs whose effect is to modify the schema of the database by adding, changing, or deleting definitions of tables or other objects. These statements can be freely mixed with other SQL statements, so the DDL is not truly a separate language.
CREATE statements

Create - To make a new database, table, index, or stored procedure.

A CREATE statement in SQL creates an object inside of a relational database management system (RDBMS). The types of objects that can be created depends on which RDBMS is being used, but most support the creation of tables, indexes, users[clarify], synonyms and databases. Some systems (such as PostgreSQL) allow CREATE, and other DDL commands, inside of a transaction and thus they may be rolled back.

CREATE TABLE statement

A commonly used CREATE command is the CREATE TABLE command. The typical usage is:
CREATE [TEMPORARY] TABLE [table name] ([column definitions]) [table parameters].
column definitions: A comma-separated list consisting of any of the following
Column definition: [column name] [data type] {NULL | NOT NULL} {column options}
Primary key definition: PRIMARY KEY ([comma separated column list])
Constraints: {CONSTRAINT} [constraint definition]
RDBMS specific functionality
For example, the command to create a table named employees with a few sample columns would be:

```
CREATE TABLE employees (
    id            INTEGER      PRIMARY KEY,
    first_name    VARCHAR(50)  NULL,
    last_name     VARCHAR(75)  NOT NULL,
    dateofbirth   DATE         NULL
);
```

Note that some forms of CREATE TABLE DDL may incorporate DML (data manipulation language)-like constructs as well, such as the CREATE TABLE AS SELECT (CTAS) syntax of SQL.

DROP statements

Drop - To destroy an existing database, table, index, or view.

A DROP statement in SQL removes an object from a relational database management system (RDBMS). The types of objects that can be dropped depends on which RDBMS is being used, but most support the dropping of tables, users, and databases. Some systems (such as PostgreSQL) allow DROP and other DDL commands to occur inside of a transaction and thus be rolled back. The typical usage is simply:

DROP objecttype objectname.

For example, the command to drop a table named employees would be:

```
DROP TABLE employees;
```

The DROP statement is distinct from the DELETE and TRUNCATE statements, in that DELETE and TRUNCATE do not remove the table itself. For example, a DELETE statement might delete some (or all) data from a table while leaving the table itself in the database, whereas a DROP statement would remove the entire table from the database.

ALTER statements

Alter - To modify an existing database object.

An ALTER statement in SQL changes the properties of an object inside of a relational database management system (RDBMS). The types of objects that can be altered depends on which

RDBMS is being used. The typical usage is:

ALTER objecttype objectname parameters.

For example, the command to add (then remove) a column named bubbles for an existing table named sink would be:

```
ALTER TABLE sink ADD bubbles INTEGER;
ALTER TABLE sink DROP COLUMN bubbles;
```

Referential integrity statements
Finally, another kind of DDL sentence in SQL is one used to define referential integrity relationships, usually implemented as primary key and foreign key tags in some columns of the tables.

These two statements can be included inside a CREATE TABLE or an ALTER TABLE sentence.

2.2 Storing Rows in a Key/Value Pair

Rows can be stored in a key value pair by using naming conventions. Consider a tables:

Table: user

Id	Name	RollNum	Marks
1	Shamail Tayyab	47	78
2	Joy	22	56
3	David	19	80
4	Nick	67	73

The Key/Value equivalent of this table would be:

Key	Value
user:1	true
user:1:id	1
user:1:name	Shamail Tayyab
user:1:rollnum	47
user:1:marks	78
user:2	true
user:2:id	2
user:2:name	Joy
user:2:rollnum	22
user:2:marks	56
user:3	true
user:3:id	3
user:3:name	David
user:3:rollnum	19
user:3:marks	80
user:4	true
user:4:id	4
user:4:name	Nick
user:4:rollnum	67

user:4:marks	73

As shown in the tables, the convention used here is:

```
table_name : primary_key : column name
```

CHAPTER 3: FOREIGN KEY RELATIONSHIPS

3.1 Foreign Key Relations

In the context of relational databases, a foreign key is a referential constraint between two tables. A foreign key is a field in a relational table that matches a candidate key of another table. The foreign key can be used to cross-reference tables.

For example, say we have two tables, a CUSTOMER table that includes all customer data, and an ORDER table that includes all customer orders. The intention here is that all orders must be associated with a customer that is already in the CUSTOMER table. To do this, we will place a foreign key in the ORDER table and have it relate to the primary key of the CUSTOMER table.
The foreign key identifies a column or set of columns in one (referencing or child) table that refers to a column or set of columns in another (referenced or parent) table. The columns in the child table must reference the columns of the primary key or other superkey in the parent table. The values in one row of the referencing columns must occur in a single row in the parent table. Thus, a row in the child table cannot contain values that don't exist in the parent table (except potentially NULL). This way references can be made to link information together and it is an essential part of database normalization. Multiple rows in the child table may refer to the same row in the parent table. Most of the time, it reflects the one (parent table or referenced table) to many (child table, or referencing table) relationship.

The child and parent table may be the same table, i.e. the foreign key refers back to the same table. Such a foreign key is known in SQL:2003 as a self-referencing or recursive foreign key.
A table may have multiple foreign keys, and each foreign key can have a different parent table. Each foreign key is enforced independently by the database system. Therefore, cascading relationships between tables can be established using foreign keys.

Improper foreign key/primary key relationships or not enforcing those relationships are often the source of many database and data modeling problems.

3.2 Defining Foreign Keys

Foreign keys are defined in the ISO SQL Standard, through a FOREIGN KEY constraint. The syntax to add such a constraint to an existing table is defined in SQL:2003 as shown below. Omitting the column list in the REFERENCES clause implies that the foreign key shall reference the primary key of the referenced table.

```
ALTER TABLE <TABLE identifier>
   ADD [ CONSTRAINT <CONSTRAINT identifier> ]
      FOREIGN KEY ( <COLUMN expression> {, <COLUMN expression>}... )
      REFERENCES <TABLE identifier> [ ( <COLUMN expression> {, <COLUMN
expression>}... ) ]
      [ ON UPDATE <referential action> ]
      [ ON DELETE <referential action> ]
```

Likewise, foreign keys can be defined as part of the CREATE TABLE SQL statement.

```
CREATE TABLE TABLE_NAME (
   id    INTEGER  PRIMARY KEY,
   col2  CHARACTER VARYING(20),
```

```
   col3  INTEGER,
   ...
   FOREIGN KEY(col3)
      REFERENCES other_table(key_col) ON DELETE CASCADE,
   ... )
```

If the foreign key is a single column only, the column can be marked as such using the following syntax:

```
CREATE TABLE TABLE_NAME (
   id    INTEGER  PRIMARY KEY,
   col2  CHARACTER VARYING(20),
   col3  INTEGER REFERENCES other_table(column_name),
   ... )
```

3.2 Storing Foreign Key Relationships

Foreign Key Relations can be stored using lazy lookups and using conventions.

Consider 2 tables as:

Table: student

Id	Name	Sport
1	Shamail	1
2	Joy	3

Table: sport

Id	Name
1	Cricket
2	Badminton
3	Football

The equivalent of this in Key/Value store would look like:

sport:1	true
sport:1:name	Cricket
sport:2	true
sport:2:name	Badminton
sport:3	true
sport:3:name	Football
student:1	true
student:1:name	Shamail
student:1:sport	lookup:sport:1
student:2	true

`student:2:name`	`Joy`
`student:2:sport`	`lookup:sport:3`

As depicted, the convention is to store two tables using the method described in “Simulating Rows” and then while representing a foreign key object, use:

lookup : table_name : foreign_key

CHAPTER 4: ONE TO MANY RELATIONSHIPS

4.1 Introduction

In a one-to-many relationship, each row in the related to table can be related to many rows in the relating table. This allows frequently used information to be saved only once in a table and referenced many times in all other tables. In a one-to-many relationship between Table A and Table B, each row in Table A is linked to 0, 1 or many rows in Table B. The number of rows in Table A is almost always less than the number of rows in Table B.

To illustrate the one-to-many relationship consider the sample table design and data below:

```
authors table
============
author_id (primary key)
lastname
firstname
book_id  (foreign key - link to book_id of books table)


books table
===========
book_id (primary key)
title


author_id  lastname  firstname   ->     book_id  title
---------  --------  ---------   ->     -------  ------
0001       henry     john        ->     0001     a database primer
                                        0002     building datawarehouse
                                        0003     teach yourself sql
0002       johnson   mary        ->     0004     101 exotic recipes
0003       bailey    harry       ->     0005     visiting europe
0004       smith     adam
```

Notice that each row in the authors table is related to 0, 1 or many rows in the books table. This makes intuitive sense because an author can write 0, 1 or more than 1 books. In our example above, John Henry has written 3 books, Mary Johnson has written 1 book, Harry Bailey has written 1 book and Adam Smith has not written any books.

If you notice carefully, the above relationship between the authors table and the books table is a one-to-many relationship. Turning around, the relationship between the books table and the authors table is a many-to-one relationship.

4.2 Storing One to Many Relationships

Consider a relationship represented as:

Table: user

Id	Name
1	Shamail
2	Joy

Table: sport

Id	Name
1	Cricket
2	Hockey
3	Badminton

Table: plays

Id	user	sport
1	1	1
2	1	2
3	2	1
4	2	3
5	2	2

The same can be represented in Key/Value store as:

user:1	true
user:1:id	1
user:1:name	Shamail
user:2	true
user:2:id	2
user:2:name	Joy
sport:1	true
sport:1:id	1
sport:1:name	Cricket
sport:2	true
sport:2:id	2
sport:2:name	Hockey
sport:3	true
sport:3:id	3

sport:3:name	Badminton
plays:1	true
plays:1:id	1
plays:1:user	lookup:user:1
plays:1:sport	lookup:sport:1
plays:2	true
plays:2:id	2
plays:2:user	lookup:user:1
plays:2:sport	lookup:sport:2
plays:3	true
plays:3:id	3
plays:3:user	lookup:user:2
plays:3:sport	lookup:sport:1
plays:4	true
plays:4:id	4
plays:4:user	lookup:user:2
plays:4:sport	lookup:sport:3
plays:5	true
plays:5:id	5
plays:5:user	lookup:user:2
plays:5:sport	lookup:sport:2

Here, we are using an intermediary table to store the relation and then using “Simulating Rows” to represent that relation.

CHAPTER 5: MANY TO MANY RELATIONSHIPS

5.1 Introduction

In systems analysis, a many-to-many relationship is a type of cardinality that refers to the relationship between two entities (see also entity–relationship model) A and B in which A may contain a parent row[clarify] for which there are many children[clarify] in B and vice versa. For instance, think of A as Authors, and B as Books. An Author can write several Books, and a Book can be written by several Authors. Because most database management systems only support one-to-many relationships, it is necessary to implement such relationships physically via a third junction table (also called cross-reference table), say, AB with two one-to-many relationships A -> AB and B -> AB. In this case the logical primary key for AB is formed from the two foreign keys (i.e. copies of the primary keys of A and B).

In web application frameworks such as CakePHP and Ruby on Rails, a many-to-many relationship between database tables in a model is sometimes referred to as a HasAndBelongsToMany (HABTM) relationship.

4.2 Storing Many to Many Relationships

Consider a relationship represented as:

Table: user

Id	Name
1	Shamail
2	Joy

Table: sport

Id	Name
1	Cricket
2	Hockey
3	Badminton

Table: plays

Id	user	sport
1	1	1
2	1	2
3	2	1
4	2	3
5	2	2

The same can be represented in Key/Value store as:

user:1	true
user:1:id	1
user:1:name	Shamail
user:2	true
user:2:id	2
user:2:name	Joy
sport:1	true
sport:1:id	1
sport:1:name	Cricket
sport:2	true
sport:2:id	2
sport:2:name	Hockey
sport:3	true
sport:3:id	3
sport:3:name	Badminton
plays:1	true
plays:1:id	1
plays:1:user	lookup:user:1
plays:1:sport	lookup:sport:1
plays:2	true
plays:2:id	2
plays:2:user	lookup:user:1
plays:2:sport	lookup:sport:2
plays:3	true
plays:3:id	3
plays:3:user	lookup:user:2
plays:3:sport	lookup:sport:1
plays:4	true
plays:4:id	4
plays:4:user	lookup:user:2
plays:4:sport	lookup:sport:3
plays:5	true
plays:5:id	5
plays:5:user	lookup:user:2
plays:5:sport	lookup:sport:2

Here, we are using an intermediary table to store the relation and then using “Simulating Rows” to represent that relation.

CHAPTER 6: SEARCHING/INDEXING (WHERE field='value')

6.1 Introduction

Searching can either be done using iteration, or using indexing.

6.2 Indexing

A database index is a data structure that improves the speed of data retrieval operations on a database table at the cost of slower writes and the use of more storage space. Indices can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access of ordered records.

In a relational database, an index is a copy of one part of a table. Some databases extend the power of indexing by allowing indices to be created on functions or expressions. For example, an index could be created on upper(last_name), which would only store the upper case versions of the last_name field in the index. Another option sometimes supported is the use of "filtered" indices, where index entries are created only for those records that satisfy some conditional expression. A further aspect of flexibility is to permit indexing on user-defined functions, as well as expressions formed from an assortment of built-in functions.

Most database software includes indexing technology that enables sub-linear time lookup to improve performance, as linear search is inefficient for large databases.

Suppose a data store contains N data objects, and it is desired to retrieve one of them based on the value of one of the object's fields. A naive implementation would retrieve and examine each object until a match was found. A successful lookup would retrieve half the objects on average; an unsuccessful lookup all of them for each attempt. This means that the number of operations in the worst case is O(N) or linear time. Since data stores commonly contain millions of objects and since lookup is a common operation, it is often desirable to improve on this performance.
An index is any data structure that improves the performance of lookup. There are many different data structures used for this purpose, and in fact a substantial proportion of the field of Computer Science is devoted to the design and analysis of index data structures. There are complex design trade-offs involving lookup performance, index size, and index update performance. Many index designs exhibit logarithmic (O(log(N))) lookup performance and in some applications it is possible to achieve flat (O(1)) performance.

Indices are used to police database constraints, such as UNIQUE, EXCLUSION, PRIMARY KEY and FOREIGN KEY. An index may be declared as UNIQUE which creates an implicit constraint on the underlying table. Database systems usually implicitly create an index on a set of columns declared PRIMARY KEY, and some are capable of using an already existing index to police this constraint. Many database systems require that both referencing and referenced sets of columns in a FOREIGN KEY constraint are indexed, thus improving performance of inserts, updates and deletes to the tables participating in the constraint.

Some database systems support EXCLUSION constraint which ensures that for a newly inserted or updated record a certain predicate would hold for no other record. This may be used to implement a UNIQUE constraint (with equality predicate) or more complex constraints, like ensuring that no overlapping time ranges or no intersecting geometry objects would be stored in the table. An index supporting fast searching for records satisfying the predicate is required to

police such a constraint.

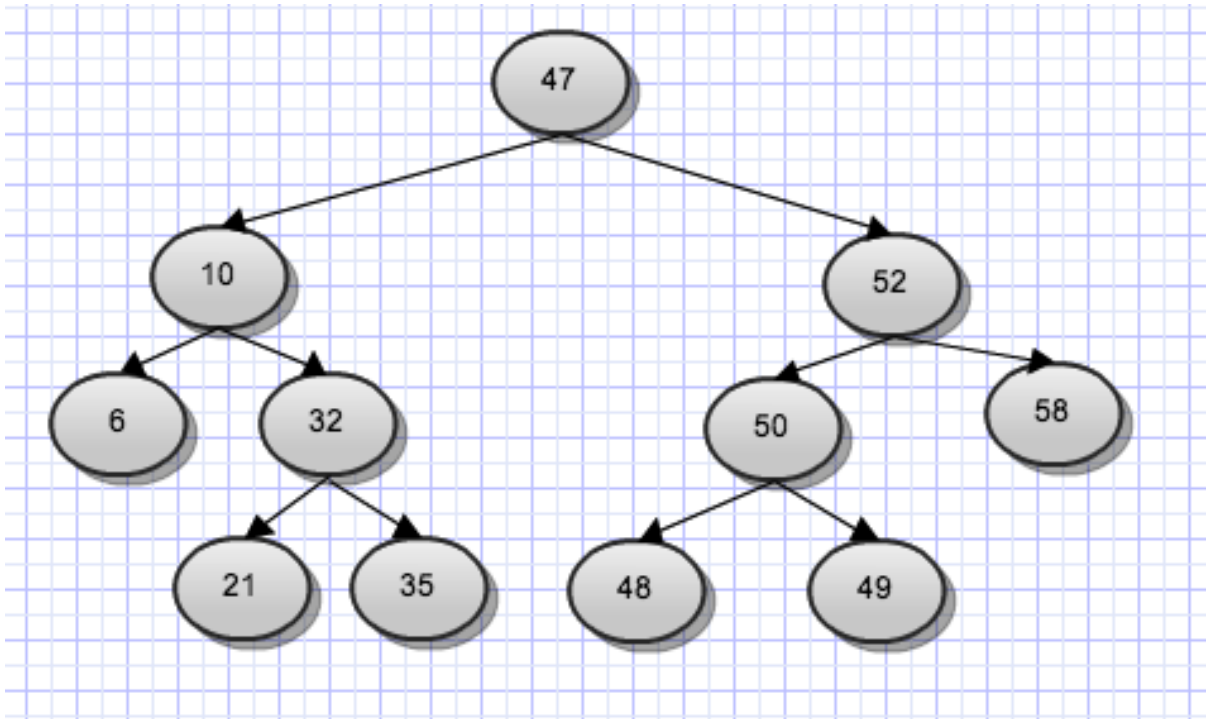
6.3 Storing Index in Key/Value

If somehow, we can store a BST in a Key/Value, then we can create an index for that column.

Consider a Table

Id	Name	RollNum
1	Shamail	47
2	Joy	10
3	David	52
4	Priya	32
5	Suresh	6
6	Geeta	58
7	Neha	50
8	Raj	48
9	Rohan	49
10	Soni	35
11	Arvind	21

If we create a BST index on RollNum column, it'll look like:



The same BST (with name index) can be stored in a Key/Value store as:

Key	Value
index	type:bst
index:0	true
index:0:value	47
index:0:left	index:1
index:0:right	index:2
index:1	true
index:1:value	10
index:1:left	index:3
index:1:right	index:4
index:2	true
index:2:value	52
index:2:left	index:5
index:2:right	index:6
index:3	true
index:3:value	6
index:3:left	null
index:3:right	null
index:4	true
index:4:value	32
index:4:left	index:7
index:4:right	index:8
index:5	true
index:5:value	50
index:5:left	index:9
index:5:right	index:10
index:6	true
index:6:value	58
index:6:left	null
index:6:right	null
index:7	true
index:7:value	21
index:7:left	null
index:7:right	null
index:8	true
index:8:value	35

index:8:left	null
index:8:right	null
index:9	true
index:9:value	48
index:9:left	null
index:9:right	null
index:10	true
index:10:value	49
index:10:left	null
index:10:right	null

CHAPTER 7: PYTHON SPECIFIC IMPLEMENTATION

The above concepts are implemented as Python in the form of an Object Relational Mapper (ORM).

```
#! /usr/bin/env python

# ------------------------------------------------------------------
# @author: Shamail Tayyab
# @date: Thu Apr 4 12:33:00 IST 2013
#
# @desc: Redis/Python ORM for storing relational data in redis.
# ------------------------------------------------------------------

import inspect
import redis

r = redis.Redis ()
r.flushall ()

class classproperty(object):
    """
    Lets support for making a property on a class.
    """

    def __init__(self, getter):
        self._getter = getter

    def __get__(self, instance, owner):
        return self._getter(owner)

class RField ():
    """
    This class defined a field in Redis Database, similar to a column
in a Relational DB.
    """

     # If this field is mandatory.
    required = False
     # The default value of this field, if not provided.
    default = None

    def __init__ (self, *k, **kw):
        if kw.has_key ("required"):
            self.required = kw['required']
        if kw.has_key ("default"):
            self.default = kw['default']

class StringField (RField):
    """
    @inherit RField
```

```
    Implementation of String Field, where user wants to store a String
in the Database.
    """
    pass

class IntField (RField):
    """
    @inherit RField
    Implementation of Integer Field, where user wants to store a
Integer in the Database.
    """
    pass

class ForeignKey (RField):
    """
    @inherit RField
    Implementation of Foreign Key, where user wants to store One to
One relation in the Database.
    """

    def __init__ (self, *k, **kw):
        RField.__init__ (self, *k, **kw)
        self.relation = k[0]

class RModel (object):
    """
    The actual Redis based model class implementation.
    """

    id = IntField ()

    keyvals = {}
    locals = []

    def __init__ (self, *k, **kw):
        """
        Stores the provided values.
        """
        self.newobj = True
        self.keyvals = {}
        self.locals = []
        self.reinit ()
        for i in self.locals:
            fieldobj = object.__getattribute__(self, i)
            if kw.has_key (i):
                self.keyvals[i] = kw[i]
            else:
                if fieldobj.required == True:
                    if fieldobj.default is not None:
```

```
                        self.keyvals[i] = fieldobj.default
                    else:
                        raise Exception ("Need a default value for %s"
% (i))

    def from_id (self, id):
        """
        Loads a model from its ID.
        """
        self.seq = int(id)
        self.newobj = False
        return self

    def reinit (self):
        """
        Reloads the properties of this class from Database.
        """
        #for name, obj in inspect.getmembers (self):
        ##    if isinstance (obj, RField):
        #        self.keyvals[name] = obj.default
        inspect.getmembers (self)

    def validate (self):
        """
        Validations for a Field.
        """
        if kw.has_key (name):
            self.keyvals[name] = kw[name]
        elif obj.default is not None:
            self.keyvals[name] = obj.default
        else:
            if obj.required:
                raise AttributeError ("This field is required")

    @property
    def classkey (self):
        """
        Generates the Key for this class.
        """
        return 'rmodel:%s' % (self.__class__.__name__.lower ())

    def sequence (self):
        """
        Sequence Generator, uses Redis's atomic operation.
        """
        seq_av_at = "%s:__seq__" % (self.classkey)
        seq = r.incr (seq_av_at)
        return seq
```

```
def prepare_key (self, key, for_seq):
    """
    Prepares a key to be stored for this class.
    """
    r_key = "%s:%d:%s" % (self.classkey, for_seq, key)
    return r_key

def save (self):
    """
    Persist this object into the Redis Database.
    """
    if self.newobj:
        using_sequence = self.sequence ()
        self.keyvals['id'] = using_sequence
        self.seq = using_sequence
    else:
        using_sequence = self.seq
    for key, val in self.keyvals.items ():
        r_key = self.prepare_key (key, using_sequence)
        r.set (r_key, val)
    self.keyvals = {}
    self.newobj = False

@classproperty
def objects (self):
    """
    Supports UserClass.objects.all () like stuff.
    """
    return InternalObjectList (self)

def __getattribute__ (self, attr):
    """
    Getter for this class.
    """
    attrib = object.__getattribute__(self, attr)
    if not isinstance (attrib, RField):
        return attrib
    if attr not in self.locals:
        self.locals.append (attr)
    if self.newobj:
        if self.keyvals.has_key (attr):
            return self.keyvals[attr]
        else:
            fieldobj = object.__getattribute__(self, attr)
            return fieldobj.default

    answer = r.get (self.prepare_key (attr, self.seq))
    fieldobj = object.__getattribute__(self, attr)
    if answer == None:
```

```
            answer = fieldobj.default
        else:
            if isinstance (fieldobj, ForeignKey):
                fkey = r.get (self.prepare_key ('__relationfor__',
self.seq))
                cls = globals ()[fkey]
                return cls.objects.get (id = answer)

        return answer

    def __setattr__ (self, attr, val):
        """
        Setter for this class.
        """
        try:
            attrib = object.__getattribute__(self, attr)
        except AttributeError:
            object.__setattr__ (self, attr, val)
            return

        if not isinstance (attrib, RField):
            object.__setattr__ (self, attr, val)
            return

        if isinstance (attrib, ForeignKey):
            self.keyvals[attr] = val.id
            self.keyvals['__relationfor__'] = attrib.relation
        else:
            self.keyvals[attr] = val


class InternalObjectList (object):
    """
    The query object, to support UserClass.objects.get () or
UserClass.object.get_by_id () etc.
    """

    def __init__ (self, classfor):
        self.classfor = classfor

    def get_by_id (self, id):
        """
        Returns an object by its ID.
        """
        clsfor_obj = self.classfor()
        clsfor_obj.from_id (id)
        return clsfor_obj
        return
        for name, obj in inspect.getmembers (clsfor_obj):
```

```
            if isinstance (obj, RField):
                key = clsfor_obj.prepare_key (name, int(id))

    def get (self, *k, **kw):
        """
        Returns an object by one of its property, say name.
        """
        if kw.has_key ('id'):
            return self.get_by_id (kw['id'])

if __name__ == "__main__":

    # Lets define a Profile Class which is a Redis Based Model
(inherits RModel).
    class Profile (RModel):
        fbid = StringField ()

    # Again, lets define a User.
    class User (RModel):
           # A Field that can store a String.
        username = StringField (required = True)
        first_name = StringField (required = True)
        last_name = StringField ()
        password = StringField (required = True)
        email = StringField (required = True)

    # Lets now define a Table which will act as foreign key for
another table.
    class FK (RModel):
           # Can store a String.
        name = StringField ()

    # Lets now define another Table Test that will have a property for
ForeignKey
    class Test (RModel):
        username = StringField ()
           # Stores a String
        password = StringField ()
           # Refers to another Table called 'FK'.
        rel = ForeignKey ('FK')
           # Stores a String with some default value.
        defa = StringField (default = 'a')
           # Stores a String with some validation.
        req = StringField (required = True, default = 'abc')

    # Creates an object of FK
    fk = FK (name = 'abc')
    fk.save ()
```

```
# See if the object is creates?
print "FKID:", fk.id

# Lets now create an object for Test
t = Test (username = "u", password = "p")
# Put the previous object as its relation reference.
t.rel = fk
# Save it.
t.save ()
print t.id

# See what we get back is the object itself!!
k= t.rel
print "Name:", k.name

#t.username = "new"
#t.save ()

#t = Test ()
#t.username = 22

# Lets see what keys were saved in the DB.
for i in r.keys ():
   print i, r.get (i)
```

CHAPTER 7: CONCLUSION

In this dissertation, the concepts are formulated first, on how to do the most common tasks in the Relational Databases in Key/Value store, then the same is implemented using Redis (which is a Key Value Store Database), and Python as an Object Document Mapper.

The source code of this project is attached with the dissertation, as well as available online at

https://github.com/pleomax00/relationalredis

Advantages:

Broadly classified, this dissertation brings effectiveness on how key/value databases can replace a traditional SQL engine, and how you can leverage it to store the data in the same way still getting all the scalability advantages that key/value store offer.

Using these approaches, the learning curve is minimized, which saves a lot of man hours in the development cycle.

The dynamic nature of key/value stores add scalability advantange, that can be seen using sharding and replication.

Disadvantages:

Although, the concepts bring in some very interesting advantages, however, it also has some disadvantages, which mostly reflects in terms of the CPU cost that user has to pay to have an object document wrapper. Although its very minimal, considering other overheads in a typical web application, such as replication, sharding, request/response overhead etc.

Project hierarchy looks like:

```
.
├── README.md
├── docs
│   ├── report.pdf
│   ├── synopsis.odt
│   ├── synopsis.pdf
│   ├── thesis.odt
│   └── thesis.pdf
├── presentation
│   ├── README.md
│   ├── apple-touch-icon.png
│   ├── css
│   │   └── impress-demo.css
│   ├── favicon.png
│   ├── index.html
│   ├── js
│   │   └── impress.js
│   └── upload.html
└── src
    └── redismodels.py
```

CHAPTER 8: Future Work

Main Stream Databases support more operations than implemented in this research, e.g

- Composite Keys
- Muti Key Indexes
- Outer and Inner Joins
- SQL syntax etc.

In future, these concepts can also be modeled in key/value stores, so that it will give more flexibility to the programmers to express their data in the most efficient manner.

CHAPTER 9: Limitations

The patterns are limited to only basic set of operations that are supported in RDBMS, and does not support any advanced operation mentioned in “Future Work” section, Also, the current implementation of the patterns is done using Redis, which is one of the key/value stores, and so forth, it does not run on any other key/value server, e.g Amazon Dynamo DB or Memcached.

The ODM (Object Document Mapper) is limited to python only, although, it can easily be ported to any other language using the same underlying principles. One third party library redis-client is required to connect with the server. Also, this implementation supports only Python 2.7, Python 3 is not (yet) supported. Git is also required to check out codebase from Github servers.