Paul Sarda

Say no to SQL depending on the circumstances

Contents

[Introduction 2](#_Toc495438819)

[Understanding NoSQL 2](#_Toc495438820)

[CAP Theorem 2](#_Toc495438821)

[BASE System 2](#_Toc495438822)

[NoSQL Categories 3](#_Toc495438823)

[Key-value stores 3](#_Toc495438824)

[Column-oriented 4](#_Toc495438825)

[Graph-Database 5](#_Toc495438826)

[Theory 5](#_Toc495438827)

[Implementation 6](#_Toc495438828)

[Document oriented 7](#_Toc495438829)

[Theory 7](#_Toc495438830)

[Implementation 8](#_Toc495438831)

[Firebase 9](#_Toc495438832)

[How data is stored 9](#_Toc495438833)

[Structuring data 10](#_Toc495438834)

[Avoid nesting data / Flat database 10](#_Toc495438835)

[Adding Data/Inserting data 11](#_Toc495438836)

[Modifying Data 12](#_Toc495438837)

[Querying Data 12](#_Toc495438838)

[Backups 12](#_Toc495438839)

[Rules 12](#_Toc495438840)

[Conclusion 12](#_Toc495438841)

[Conclusion 13](#_Toc495438842)

[Refences 14](#_Toc495438843)

# Introduction

# Understanding NoSQL

NoSQL are non-relational databases this is a fundamental difference which changes how the database is structured to its core.

# CAP Theorem

Core to all database systems is CAP theorem.

* **Consistency** – All data in the database nodes must be consistent
* **Availability** – service guaranteed with no downtime.
* **Partition Tolerance** – the database system will continue to operate even if commination between the servers is unrealable.

Achieving all three is of course impossible. Because it is about fully achieving something with complete reliability a system cannot be 100% consistent if it is

<https://www.w3resource.com/mongodb/nosql.php>

# BASE System

# NoSQL Categories

Of NoSQL databases, there are four broad categories they fit in each with their own limitations (W3resourcecom, 2017).

## Key-value stores

In a key value database, each piece of data is stored in a tuple with one being the unique key and the other being the value. The key should be a searchable value with shorter being better because it means that it can be searched more easily. While the value can be anything from a single bit to a encapsulated key value pair but this depends on the database itself (Androidcom, 2017).

The use case below is for a histogram with the key being the number and value being the number of occurrences.

|  |  |
| --- | --- |
| Key | Value |
| 1 | 10 |
| 2 | 90 |
| 3 | 30 |
| 4 | 50 |
| 5 | 20 |
| 6 | 76 |

|  |  |
| --- | --- |
| Key | Value |
| Paul | Sydney, Melbourne |
| Bandit | Bangkok, Nairobi, Beijing, Yokohama |
| Patrick | Ankara, Tokyo, Dhaka |
| Richard | Moscow, London, Tehran, Dongguan |

An implementation of a Key-Value store database is Redis

ADD MORE

## Column-oriented

In a column-oriented database the data is associated in a column rather than a row as in a traditional relational database. Unlike a relational database it makes changing existing data and accessing an entire row much more difficult. But using aggregate queries or any query which uses a single row more efficient because it only accesses the relevant row (Stavros harizopoulos, daniel abadi, peter boncz, 2009).

For example, refer to the tables below.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Row-orientated** | | |  | **Column-oriented** | | |
| Name | DOB | Country |  | Name | DOB | Country |
| Paul | 1/01/1900 | Brazil |  | Paul | 1/01/1900 | Brazil |
| Patrick | 1/02/1900 | Mongolia |  | Patrick | 1/02/1900 | Mongolia |
| John | 1/02/1901 | Austria |  | John | 1/02/1901 | Austria |

In the Row-oriented database using SELECT COUNT(Name) FROM Whatever would select 3 rows with 3 columns each while in a column-oriented database it would select 3 rows and a single column. With the additional benefit of SELECT \* FROM Whatever is the same speed because it is simply selecting every row.

ADD MORE?

## Graph-Database

### Theory

In a graph database, the data is stored in a mathematical graph using vertices and edges intend of tables, rows, columns and joins which sounds confusing and crazy because it is (W3resourcecom, 2017).

shown in figure 1 are the vertices and edges of a graph. Shown in figure 2 is a filled in graph database. As shown each of the vertices is a property with a value and a column equivalent eg Name is green, Purple is a game, Red is a location. While edges are the relationships with the nature of the relationship being labelled on the edge eg plays. So, by simply looking at the graph we find a huge amount about how the vertices are related for example we know Paul plays AOEII and DOTA2, he lives in brazil, knows Patrick and john.

The way this method of storing relationships is different from in a relational database where the relationship is stored is with foreign keys stored in a table which must exist as a primary key in another table. There is no label representing the connection outside of the ERD in the actual database. It also eliminates weak entity’s because each node can have as many edges to other nodes as it needs (Neo4jcom, 2017).

This also solves this issue in relational database of using inner joins to view relationship being costly needing to view multiple tables find the foreign keys connection. if the relationship is attached to the vertices instead each relationship can be instantly viewed.

Implementation of a graph-database is on the next page.

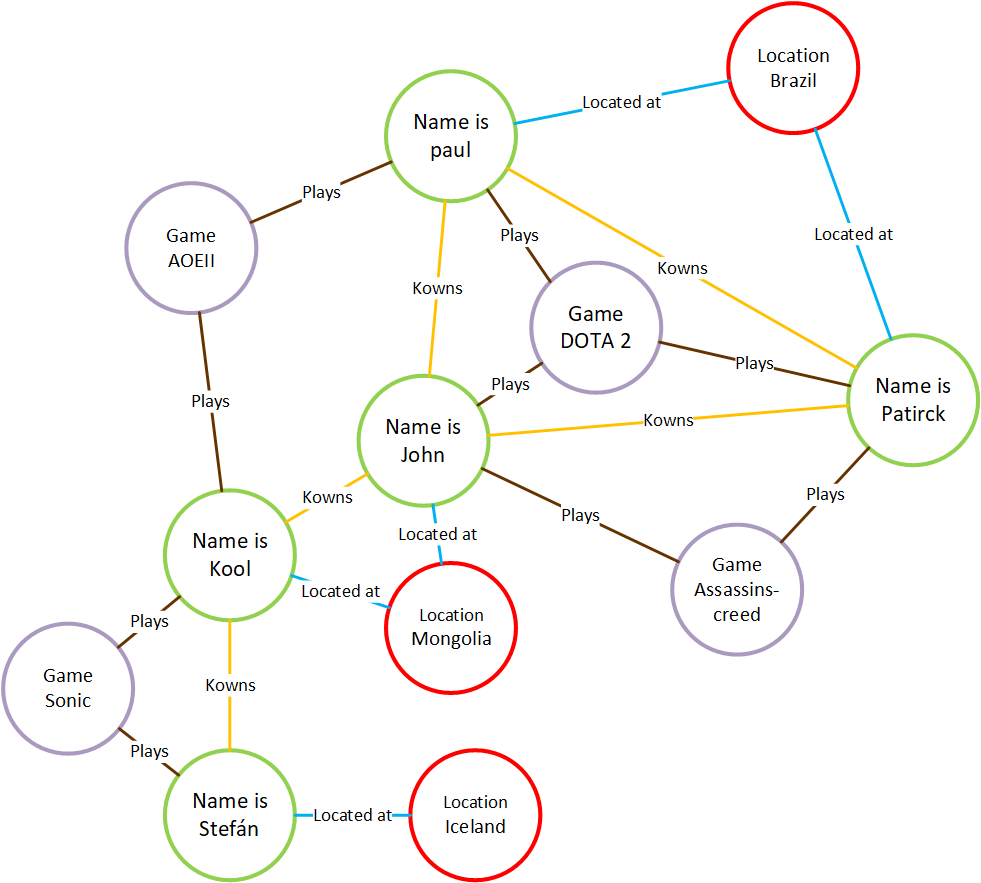


Figure 2

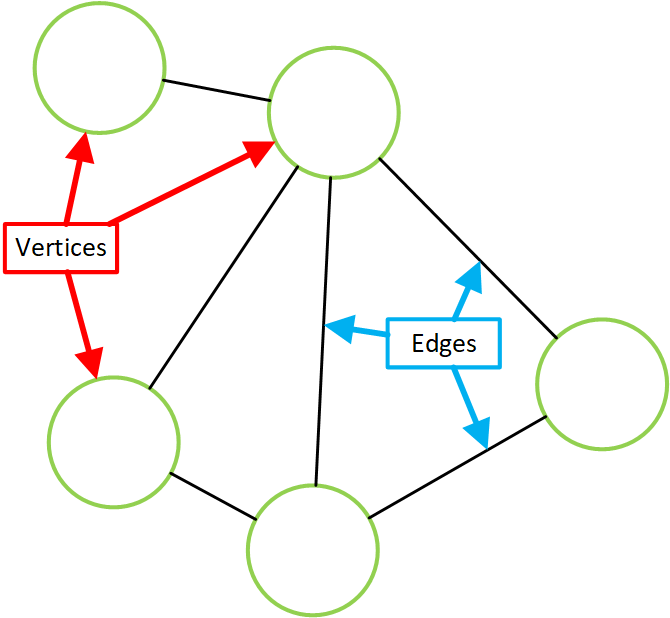


Figure 1

### Implementation

Of course, the implementation depends on which graph database is being used in this example I will be looking at neoj4 query language cypher and how the commands map to SQL.

In Cypher many sql Statements are the same eg

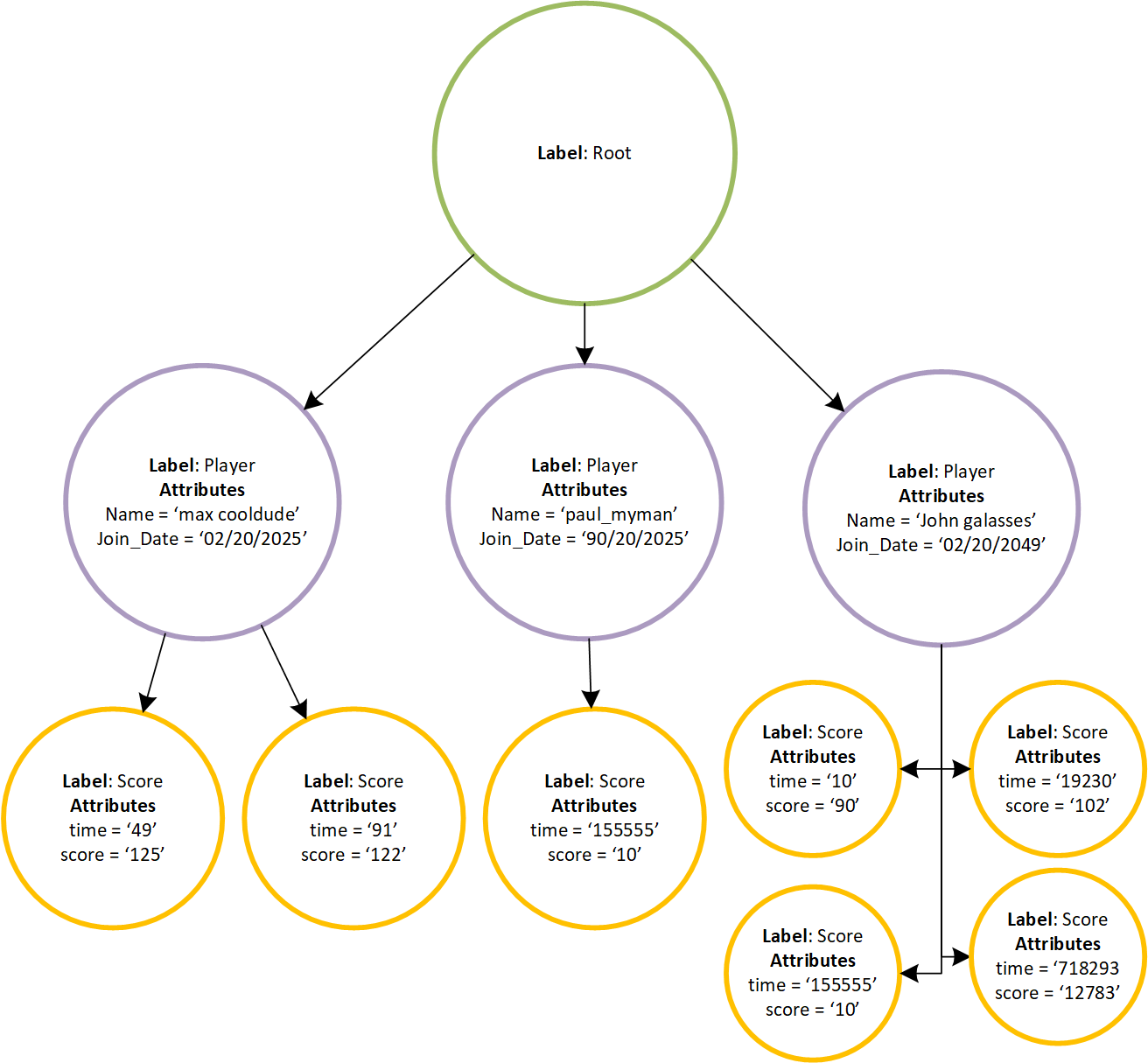
## Document oriented

### Theory

Document oriented database are centred around the A in CAP Theorem prioritising accessibility over all else. used in applications where it isn’t the end of the world if the data isn’t 100% up to the second so think a scoreboard not a banking database (lerman, 2011). Focusing on telemetering joins and strain on database servers and moving computing to the client.

How data is stored in a Document oriented database is by using a series of nodes with each node able to have many attributes and many child nodes. With the first node being the root node and every other node in the database being a child of said node (lerman, 2011).

Shown below is an example of the hierarchy.



As shown each node can accesses any other node by going up to their parent node until reaching a common parent. Along with limited restrictions including two separate nodes may have the same attribute name but there cannot be two attributes of the same name inside of the same node.

How can primary keys work? The primary key can be created by using the hierarchy and sequential number which depending on the system may be automatically generated or may be provided by the user.

### Implementation

A simple and common document oriented database is XML with a document looking like this.

<<!--- Root, Each Node which is opened must be closed --->

<Root>

<!--- Node with 2 attributes --->

<Player name='max\_coolDude' join\_date='02/05/2025'>

<!--- Nodes do not have to have children --->

<Score time='49' score='125' />

<!--- Multiple nodes of the same label can be together --->

<Score time='91' score='122' />

<!--- attributes cannot be in the closing tag --->

</Player>

<!--- Closing tag --->

</Root>

And for interpretation purposes in a relational database the ERD would look like this.



Notice how there is no need for any inner joins in the Document-oertinted version but also no type checks or enforceable rules while entering data but this is dependent on the database with the caveat that the whole point is to reduce load on the server and move it to the client.

# Firebase

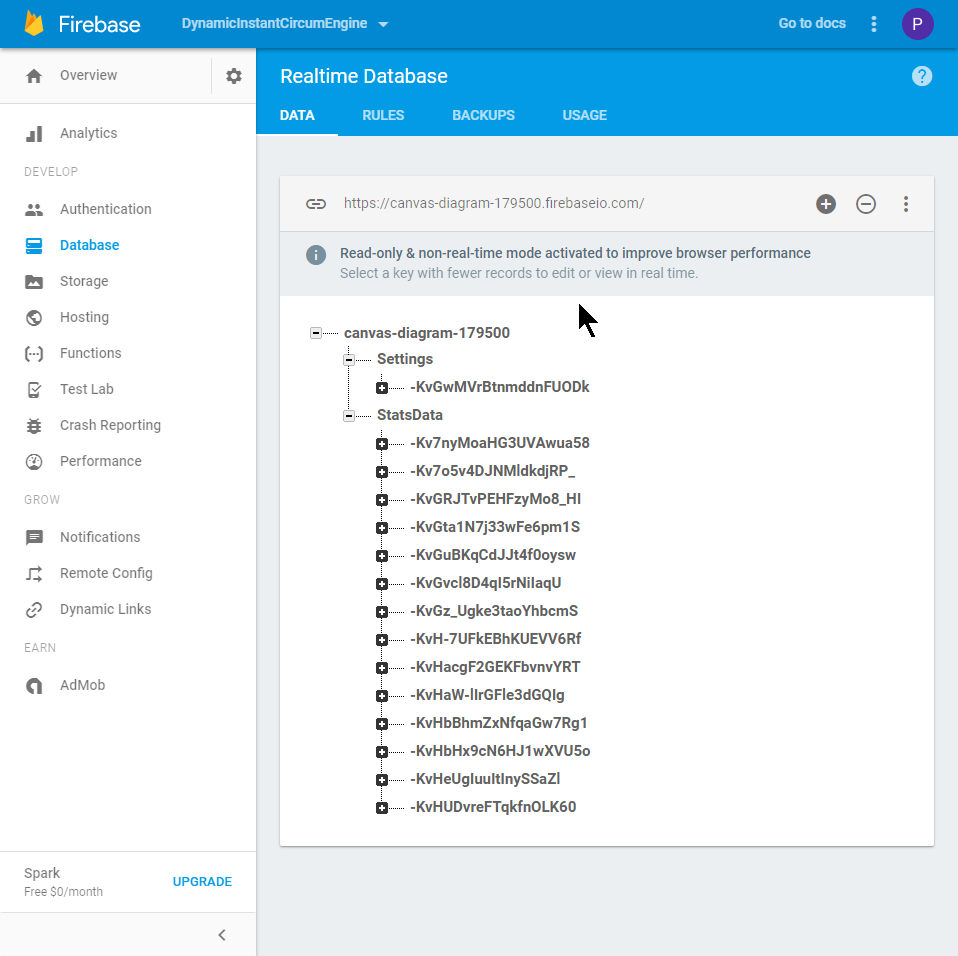
Firebase is a suite of multiplatform tools the full list can be found here <https://firebase.google.com/products/> but the focus of the report is the real-time database and the authentication is talked about in the report.

The real-time database offers an interface to a centralized online database with API’S available for IOS, Android, Admin SDK, REST API, C++ and Unity; Allowing for a single database to serve each platform. (Googlecom, 2017).

For my investigation on firebase I will be using the android API.

## How data is stored

Firebase uses a what is essentially a could hosted JSON tree holding the same structure and rules as a JSON file. Being able to store basic data types, strings, integers, doubles, Booleans, etc (Googlecom, 2017). A screenshot of the database viewed on the online console below.



As shown it is just a JSON tree really.

## Structuring data

Firebase is a Document oriented NoSQL database so I will not be reiterating the points I made here [Implementation](#_Implementation).

### Avoid nesting data / Flat database

Firebase allows nesting up to 32 levels deep from the root this is a clear instance of just because you can doesn’t mean you should reasoning why stated in the points below. When data is retrieved from the database all child nodes will be automatically also retrieved.

To access the name field each of the targets and its child rolls will also be accessed. The scope of the example is limited but there could be millions of rolls in each target all data which you don’t care about when you are accessing the name of the stats.



How can you “Flatten data” by breaking up the family and removing the mountain in this case targets to separate parent a devoice if you will. Shown below is this in practice. Now the name can be accessed in a few bytes (Googlecom, 2017).



## Adding Data/Inserting data

Adding data will depend on the platform for android firebase can store any of the following datatypes.

* String
* Long
* Double
* Boolean
* Map<String(Key), Object>: creates a nested value with the string being the key.
* List<Object>: creates a nested value with the index being the key.

To store data, you simply pass a custom java object which must have an empty constructor. Each public Getter with a Setter will be stored in the Realtime database any datatypes which don’t match the previously mentioned will be casted to the appropriate datatype eg Ints as Longs (Google, 2017).

Shown below Is my custom java class which is in Kotlin. In order to store the stats data for my application in firebase more effectively I created a specialised storage class. Flowing previously mentioned Document oriented database design.

class StatsDataStorage() {

var Name : String = ""

var Sides : Int = 0

var CreationDate : Long = 0

var AverageRoll : Double = 0.0

var UserID : String = ""

var NumberOfRolls : Long = 0

var Seed : Long = 0

var Targets = ArrayList<String>()

/\*

takes in stats data and stores

what needs to be in the database

\*/

constructor(statsData : StatsData, userId : String) : this(){

…BORING SETTERS…

}

}

That’s all fun a good but what about writing the data into the database how? With the function below comments explain what is happening and why

fun WriteNewStatsData(statsData : StatsData?){

// Gets the userID from firebase

val userId = FirebaseAuth.getInstance().currentUser?.uid

if(userId != null && statsData != null){

//Creates the storage class

val statsDataStorage = StatsDataStorage(statsData, userId)

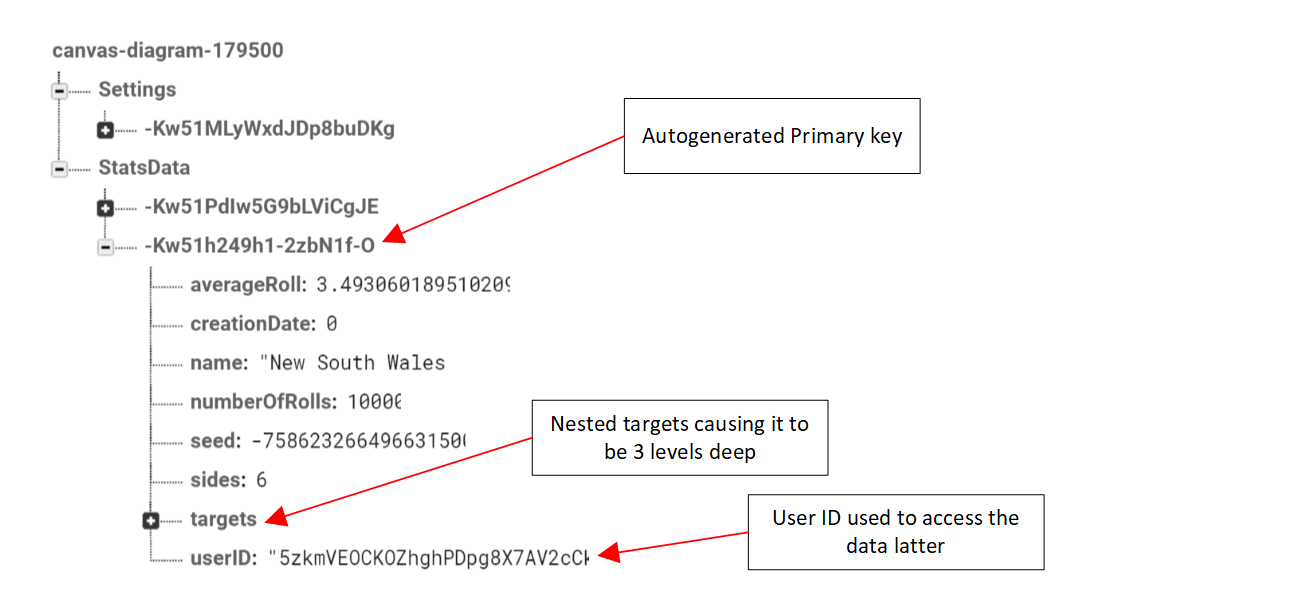
//Use push to create a autogenerated key

StatsDbRef.push().setValue(statsDataStorage)

}

}

Push() is used to create a new child with a key which must be done so the data can be queried. This is the only way without a key. Remember that accessing data and querying data are separate.

Shown below is the data inserted into the Realtime database.

## Querying Data

Querying data in firebase is done though a query object.

### Obtaining a query object

You cannot simply create a new Query() a query object must be created from a database refence which is a query because it inheritances from query. Meaning that it can call all query methods eg orderByChild() Which orders the nodes by the stated child node. A detailed table below

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Return | Action | Lose SQL equivalent |
| ref | Database Reference | Returns the database refence which can be used to modify the data as stated below. | N/A |
| endAt() | Query | will create a query containing only children where the value is less than or equal to given value. | WHERE col >= value |
| equalTo() | Query | Will create a query containing only children where the value is equal to. | WHERE col == value |
| startAt() | Query | will create a query containing only children where the value is greater than or equal to given value. | WHERE col <= value |
| orderByChildren() | Query | Will create a ordered query where all child nodes have been ordered by the given child | ORDER BY col |
| limitToFirst() | Query | Create a query that will only contain the first x children | N/A |
| limtToLast() | Query | Create a query that will only contain the last x children |  |

Once a query or database refence has been obtained with the wanted children to retrieve the data a value event listener must be created and given to a Query or Database refence Object. A data change listener is an interface which two methods onDataChange() and onCannceld(). On the next page is a example class in Kotlin.

private val loadSettings = object : ValueEventListener {

// Called when data is reteived

override fun onDataChange(dataSnapshot: DataSnapshot) {

// data Snapshot conaitns whichever level

// the databse refence was at with it's children

for (i in dataSnapshot.children) {

// Reverses the json conversion when storing

// to turn in back into a Java class

\_settingsData = i.getValue(SettingsData::class.java)

}

// if no data has been loaded

// it will be assigned as null

if(\_settingsData == null){

// Writes new data into database

\_settingsData = SettingsData()

FireBaseHelper.instance.WriteSettings(\_settingsData as SettingsData)

}

// Used by the rest of the program because this is triggered

// at an unknown time so this will trigger the code depended

// on it to load

SettingsLoaded = true

\_propertyChangeSupport.firePropertyChange(SETTINGS\_LOADED, false, true);

}

// Called if Data cannot be retievred

override fun onCancelled(databaseError: DatabaseError) {}

}

Which is assigned to a database refence with the addValueEventLisnter() or addListenerForSingleValueEvent(). With the ladder only trigger once and the former tirrger each time the database is changed. Bellow is an example.

fun loadSettingsData(){

val userID = FirebaseAuth.getInstance().currentUser!!.uid

// Gets a query with the users setting data as a child

\_curSettingsQuery = FireBaseHelper.instance.UserSettingsQuery(userID)

// Adds the event listener which will trigger whenever the data

// is modified

\_curSettingsQuery!!.addValueEventListener(loadSettings)

}

### Inner join

As you have noticed as all documents-oriented databases there are no join queries and no foreign key constraints. This is of course done to speed up delivery of data and reduce load on the server but connecting data from two separate tables will need to be done in most databases.

Below is the structure of the database where we wish to get all of the target data for a given statsData. Notice how we could simply nest the data but that would increase load times for loading a statsData Object along with wasting bandwidth loading resources which we don’t need at the time. Which is why it's in a separate child in the root node



Each Target Data child has Its owners key keep in mind these are autogenerated by firebase making the process more complex but still not too confusing bellow on the next page in the implementation in android

private val LoadStatData = object : ValueEventListener {

override fun onDataChange(dataSnapshot: DataSnapshot) {

var statsDatastorage : StatsDataStorage? = null

var statsDataKey : String = ""

// loads the object from firebase

for (i in dataSnapshot.children) {

statsDataKey = i.key

statsDatastorage = i.getValue(StatsDataStorage::class.java)

}

// gets the target storeage from the targetData table

private val LoadTargets = object : ValueEventListener {

override fun onDataChange(dataSnapshot: DataSnapshot) {

var targetsStorage : TargetsStorage? = null

for (i in dataSnapshot.children){

targetsStorage = i.getValue(TargetsStorage::class.java)

}

// Creates the statsData object with both data classes

val statsData = StatsData(statsDatastorage, targetsStorage)

SelectedData = statsData

}

override fun onCancelled(databaseError: DatabaseError) {}

}

if(statsDatastorage != null && statsDatastorage.Name == name){

// Applies the next listner to query the databse

FireBaseHelper.instance

.TargetDataForStats(statsDataKey)

.addListenerForSingleValueEvent(targetLisiner)

}

}

override fun onCancelled(databaseError: DatabaseError) {}

}

public fun LoadStatsDataForKey(key : String){

val userID = FirebaseAuth.getInstance().currentUser!!.uid

// Returns a query with all statsData for a

// Single Users

val allStatsForUser = FireBaseHelper

.instance.AllStatsDataForUser(userID).ref

// applies the lisnter to the data in the query

allStatsForUser

.child(key)

.addListenerForSingleValueEvent(LoadStatData)

}

### Use Case

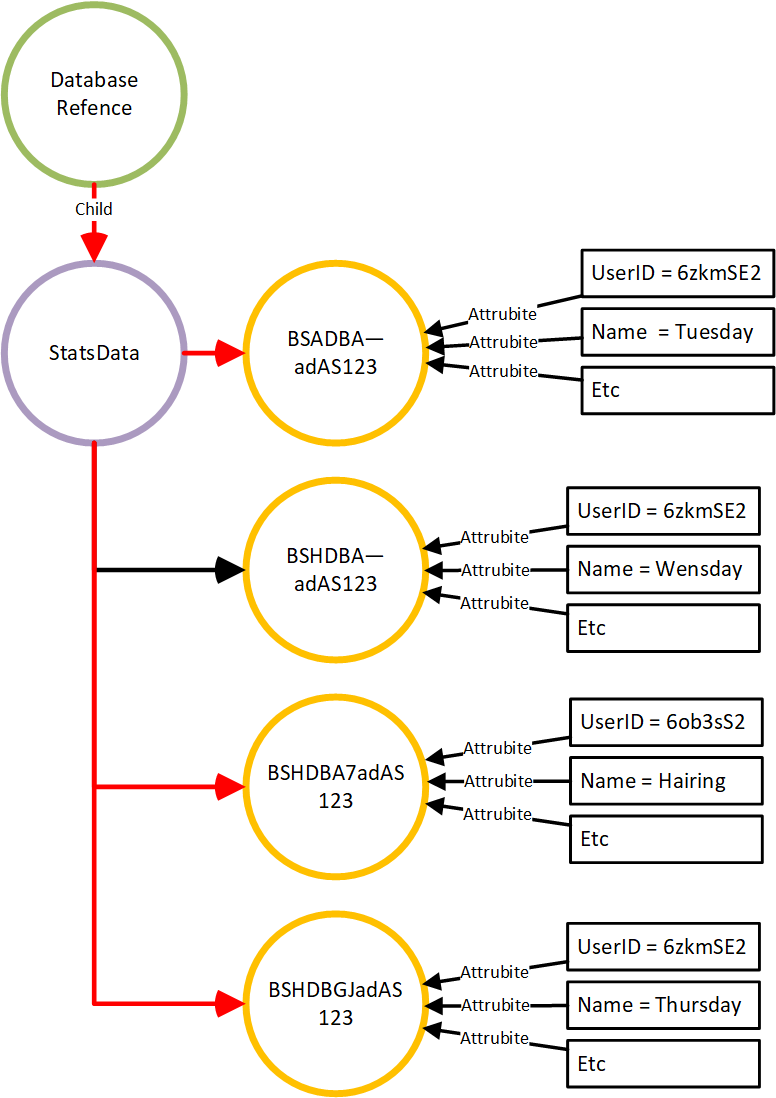
Due to firebase being a document oriented database there are no where satements or join statements which for obvious reasons makes querying data different. For example, below I need to receive each stats data for a user.

To achieve this I must start at the root node access the StatsData child then create query with the OrderByChild() method which I can select a “child” to order the results remember that because everything in firebase is at it’s core a primitive data type this is always possible. This returns a query object. Which has servual methods for querying the results below I use equalTo which as the name implies will filter all results which are not equal to the given value.

val DatabaseRoot : DatabaseReference = FirebaseDatabase.getInstance().reference

fun AllStatsDataForUser(uid : String) : Query{

return DatabaseRoot.child(STATS\_NAME).orderByChild("userID").equalTo(uid)

}

## Modifying Data

### Updating data

It is possible to update specific fields in firebase using the updateChidlren() method. This method relies on the data structure of Map<String, Object> with the string being the key and the object being the value (Google, 2017).

For example, in my mobile apps database anonymous users can save data but when they login I want to transfer all of the anonymous users saved data to the new logged in user. I could get the value for each read the entire object modify a single field then write over the old object but this seems excessive. Which is where the update children method comes in.

ADD EXAMPLE

Another alternate is to simply find the child you wish to update and just set the value again which is essentially what the Update Children method is but this is a fine alternative for only updating a single child () while the update children method can update many children at once (Google, 2017).

This is done as expected by using a Query getting the refence and setting the value again.

if(!dataSnapshot.hasChildren()){

query.ref.push().setValue(settingsData)

}else{

for (i in dataSnapshot.children) {

i.ref.setValue(settingsData)

}

}

## Backups

What good is a database without any ability to roll back to older instances or any kind of backup. There are many solutions considering that the database is at the core just a json document.

* Automated / Official solution on page 18
* Manual download on page 18
* Custom Libraries on page 18

### Automated / Official solution / Blaze

Offers an integrated backup solution which takes daily backups of the database and rules. With it costing X amount per month. Backups occur once day are a time when the server load is low. Backups are kept for 30 days

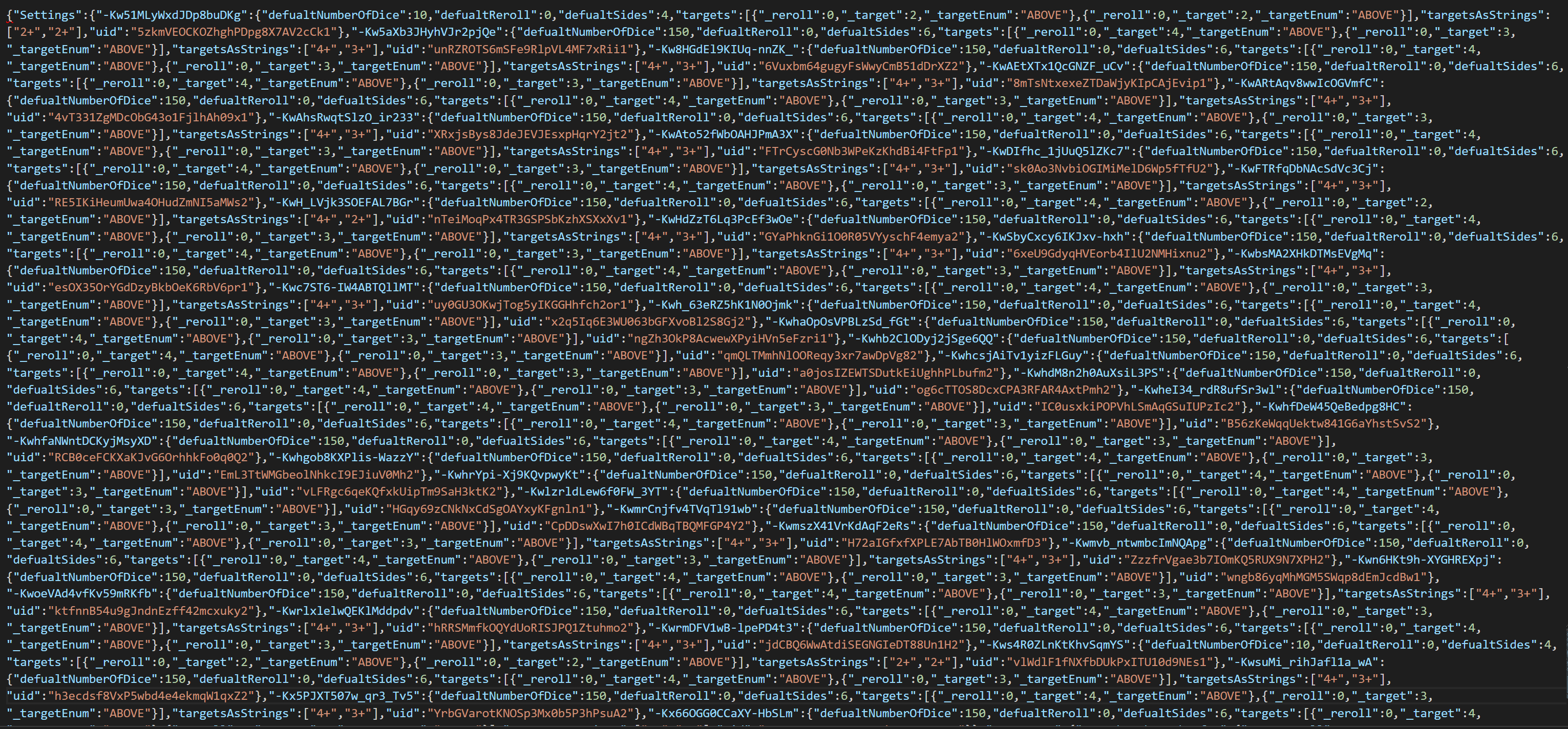
### Manual download

Because it’s a json file you can simply use

curl 'https://<FirebaseProjectName>.firebaseio.com/.json?auth=<TOKEN>

which will download the raw json file this can be slow and will download the entire library at once. Because it will just download the value at the /. It will not be made pretty unless spefised and you will get something like this which makes sense saving space on the whitespace and tabs because this nightmare is only 14KB while the pretty version is 22KB.

Shown below is the pretty version downloaded directily from the webstie but can also be obratined though curl and the ugly version im not going to bother to say which is which.



{

"Settings" : {

"-Kw51MLyWxdJDp8buDKg" : {

"defualtNumberOfDice" : 10,

"defualtReroll" : 0,

"defualtSides" : 4,

"targets" : [ {

"\_reroll" : 0,

"\_target" : 2,

"\_targetEnum" : "ABOVE"

}, {

"\_reroll" : 0,

"\_target" : 2,

"\_targetEnum" : "ABOVE"

} ],

"targetsAsStrings" : [ "2+", "2+" ],

"uid" : "5zkmVEOCKOZhghPDpg8X7AV2cCk1"

},

}…

This of course can be used to create a custom library which will automatically backup and please note this does not eat into the allocated data usage as far as I can tell.

### Custom Libraries

## Rules

## Conclusion

# Conclusion

# Refences

W3resourcecom. 2017. *W3resource.* [Online]. [24 September 2017]. Available from: <https://www.w3resource.com/mongodb/nosql.php>

Androidcom. 2017. *Androidcom.* [Online]. [26 September 2017]. Available from: <http://database.guide/what-is-a-key-value-database/>

Stavros harizopoulos, daniel abadi, peter boncz, S.H.,.D.A.,.P.B. 2009. *Yaleedu.* [Online]. [26 September 2017]. Available from: <http://www.cs.yale.edu/homes/dna/talks/Column_Store_Tutorial_VLDB09.pdf>

Neo4jcom. 2017. *Neo4j Graph Database.* [Online]. [27 September 2017]. Available from: <https://neo4j.com/developer/graph-db-vs-rdbms/>

lerman, J.L. 2011. *Data Points - What the Heck Are Document Databases?* [Online]. [28 September 2017]. Available from: <https://msdn.microsoft.com/en-us/magazine/hh547103.aspx>

Googlecom. 2017. *Firebase Realtime Database.* [Online]. [30 September 2017]. Available from: <https://firebase.google.com/docs/database/>

Googlecom. 2017. *Structure Your Database.* [Online]. [1 October 2017]. Available from: <https://firebase.google.com/docs/database/web/structure-data>

Google, G. 2017. *Read and Write Data on Android.* [Online]. [10 October 2017]. Available from: <https://firebase.google.com/docs/database/android/read-and-write>

Google, G. 2017. *Read and Write Data on Android.* [Online]. [21 October 2017]. Available from: <https://firebase.google.com/docs/database/android/read-and-write>