Have Your Data and Query It Too: From Key-Value Caching to Big Data Management

Dipti Borkar
Couchbase Inc.
dipti@couchbase.com

Ravi Mayuram
Couchbase Inc.
ravi@couchbase.com

Gerald Sangudi
Couchbase Inc.
gerald@couchbase.com

Michael Carey
Couchbase Inc.
mike.carey@couchbase.com

ABSTRACT

Couchbase Server is a rethinking of the database given the current set of realities. Memory today is much cheaper than disks were when traditional databases were designed back in the 1970's, and networks are much faster and much more reliable than ever before. Application agility is also an extremely important requirement. Today's Couchbase Server is a memory- and network-centric, shared-nothing, auto-partitioned, and distributed NoSQL database system that offers both key-based and secondary index-based data access paths as well as API- and query-based data access capabilities. This is a major change from Couchbase's roots; in its early days, its focus was entirely on high performance and highly available key-value (memcache) based caching. Customer needs and competitive pressures in the evolving non-relational database market also accelerated this change.

This paper describes the architectural changes needed to address the requirements posed by next-generation database applications. In addition, it details the implementation of such an architecture using Couchbase Server and explains the evolution of Couchbase Server from its early roots to its present form. Particular attention is paid to how today's Couchbase Server cluster architecture is influenced by the memory-first, high-performance, and scalability demands of typical customer deployments. Key features include a layer-consolidated cache, a consistency-controllable interplay between updates, indexes, and queries, and a unique "multi-dimensional" approach to cluster scaling. The paper closes with a look at future plans for supporting semi-structured operational data analytics in addition to today's more OLTP-like, front-facing use cases

1. INTRODUCTION

Today's operational or OLTP applications have very different requirements than those of the past. Flexibility, scale, availability, and performance requirements have all seen dramatic changes. With population-scale user bases, the workload and throughput requirements have grown in many cases to hundreds of thousands of reads and/or writes per second. In addition, with the need to offer rich, on-line user experiences at high throughput, latency requirements are being pushed ever lower, with 1-3 milliseconds being a common latency expectation for applications like user profile stores. Applications such as catalog and SKU management systems need the ability to change and update information on the fly. All of these requirements come with the additional need to scale elastically with demand while being always available. These requirements call for a next-generation of database systems.

This paper begins by providing some general background on NoSQL systems followed by a brief history of Couchbase. We then describe the design choices that a database system needs to make to support OLTP workloads (vs. OLAP workloads) with a focus on the core principles underlying Couchbase Server. This is followed by an in-depth look at both its external interfaces and its under the hood architecture. We close with a look into the future.

NoSQL is a term applied to a class of DBMSs that are generally designed for use in high data volume, high throughput, and web-scale applications [3, 7]. NoSQL databases are sometimes also called non-relational databases because many of the systems in this category allow nesting and have no fixed schema, thus enabling the storage of complex entities as well as semi-structured data and allowing more flexible use by developers. Most have provided non-SQL-based APIs for accessing data since they were popularized primarily through grass roots developer adoption. However, over time, some have added query-based interfaces in addition to their APIs or MapReduce-style paradigms.

Common characteristics for NoSQL databases include horizontal scalability via a clustered approach and greater schema flexibility than relational databases. Beyond these characteristics, NoSQL databases are often classified into four categories, as indicated in Figure 1. Over time, these categories are starting to converge [10], and we believe that as NoSQL systems mature further, a single platform should be capable of supporting most of the features found in these categories today.

i. Key-Value Stores

Key-value stores provide fast access to a value when its key is known. They are the foundation of NoSQL databases and are typically implemented via a distributed hash table where each entry has a unique key plus an associated data value.

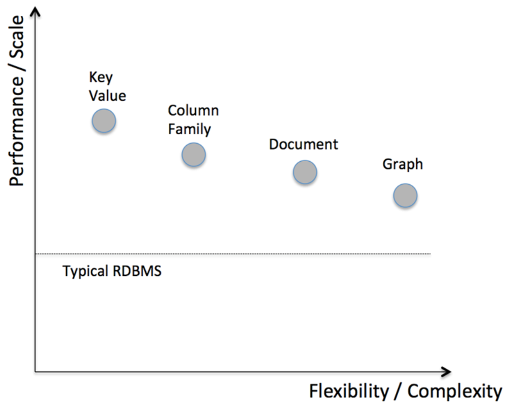

Figure 1. NoSQL database categories

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2882903.2904443

The key-value model is the simplest and the easiest to implement. It also provides the highest raw performance; applications that need massive scale along with very low latency and/or high throughput are often implemented using this approach today. If the key for a given item (value) or set of items (values) is not knowable a priori, however, key-value systems are not a particularly good fit; most of the querying and management of relationships would then have to be handled in the application tier. Examples of key-value systems include Aerospike, Redis, and DynamoDB.

ii. Column Family Databases

Column family databases provide a bit more flexibility than key-value databases. A key is still needed to access the data, but the data value itself can consist of multiple columns. Schemas are flexible, and each keyed item can have its own set of columns (distinct from those of other items). Columns are organized as column families, and some such systems provide the ability to query by more than just the key. Examples of column family databases include BigTable, HBase, and Cassandra.

iii. Document Databases

Document databases provide still greater flexibility. The primary access path in a document database is still the key, but the value is typically based on a semi-structured data format like JSON (or XML). This allows complex objects and entities to be represented. These systems also typically provide more than just key-based access. Limited querying is usually supported as well, either through an API-based approach, MapReduce-based querying, or a (SQL-like) query language. Document database examples include Couchbase, MongoDB, and MarkLogic.

iv. Graph Databases

Graph databases build on document databases by adding a flexible graph model. Data is modeled as nodes, edges and properties. Typically these databases also use a semi-structured format like JSON to represent data. APIs are common to provide access to data, both for values and for following relationships, but support for full SQL-like query capabilities is rare since their focus is largely on querying graph structures. Examples of graph database systems include Neo4j and OrientDB.

1.1 Brief Couchbase History

Couchbase, Inc. was created through the merger of Membase, Inc. and CouchOne, Inc. in 2011. This combined two technologies, Membase Server and CouchDB. Membase Server was a key-value distributed database based on memcached, while CouchDB was a single-node document database supporting JSON. The goal of the merged company was to combine the strengths of Membase as well as CouchDB to create a highly scalable, high-performance, document-oriented database capable of addressing the needs of next-generation web, mobile and IoT applications.

1.2 Membase Server

When Couchbase began, Membase Server was a distributed key-value store built on top of memcached. Memcached itself is a widely used single-node caching technology in the Internet application space. It provides consistent, low-latency access to objects. Objects are binary in nature and similar to BLOBs in databases. Membase used memcached for its integrated caching layer and offered an API similar to the memcached API for user access to data.

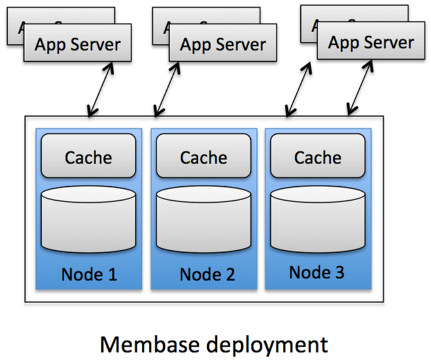

Figure 2. Membase deployment architecture

Membase added several major capabilities to the pure memcached system, including a distributed hash-partitioning scheme for scaling out, asynchronous persistence to disk to handle workload spikes at memory-first speeds, and asynchronous data replication within a cluster to achieve high availability.

Figure 2 illustrates a typical deployment of Membase for a web application. Since the caching layer is now consolidated with a persistent key-value data store, an additional external cache is no longer required for efficiently serving requests. Requests from the application servers are distributed across the collection of Membase nodes based on document hashing.

1.3 CouchDB

CouchDB was initially built as a single node database system with no partitioning capabilities and with several major differences from relational database systems. The first difference was schema flexibility. CouchDB chose JSON to serve as its data format, allowing agile application development without the need to conform to a strict schema. This difference also eliminated the need to perform complex multi-way joins just to reassemble normalized data.

The second difference from traditional relational database systems was tunable consistency between data and the associated indexes. Direct document access using a key was always consistent, but query access based on indexes was eventually consistent by default. A user could, however, specify a desired level of consistency at query submission time based on their type of data and their application's needs. CouchDB does support database-to-database replication (not partition level) to bring data closer to users. Figure 3 illustrates a CouchDB deployment with replication enabled.

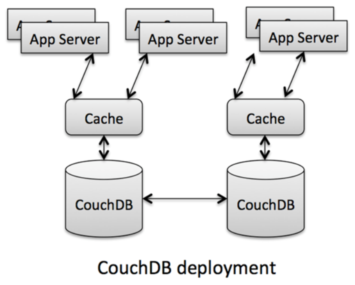

Figure 3. CouchDB deployment architecture

2. COUCHBASE SERVER OVERVIEW

2.1 Towards a Next-Generation DBMS

Couchbase set out to achieve specific technology goals, targeting new applications and their new requirements as explained earlier, through its merging of the Membase and CouchDB technologies. In general, many new applications (as well as some existing applications), including user profile services, custom content and metadata applications, and custom catalog or asset management applications, do not require a full relational database management system (RDBMS). Over time, their performance, scale, and flexibility needs outweigh their transactional requirements.

With Couchbase Server, it is not necessary to normalize natural application objects into multiple tables. As in CouchDB, objects can be stored in their unnormalized form as JSON documents. In addition, Couchbase Server allows users to access their data using either the key-value approach or a query-based approach. This means that applications that require high performance as well as the flexibility to access a sub-set of their data or parts of an object don't have to use two different systems; one database can support both of those access approaches.

RDBMs have dominated the IT industry for almost 40 years. However, NoSQL systems are now reaching a point where substantial functionality from the RDBMS world is available without the associated disadvantages of rigid schemas or lack of scaling. There are still areas, like transactionality, where further rethinking is needed regarding what is practical and needed by the next generation of applications, but the time seems right for next generation databases (such as Couchbase, of course) to transcend the relational model for many applications.

2.2 The OLTP vs. OLAP Design Tradeoff

Today, Couchbase Server is focused on handling OLTP-like workloads (i.e., OLTP workloads that do not need strong, multi-operation ACID transactionality). It offers a range of capabilities to manage and access data in real-time and allows users to isolate their workloads via a multi-dimensional scaling (MDS) approach. This will be explained in more detail in Sections 2.3.1 and 4.4. Couchbase Server can also handle limited OLAP-style workloads using pre-computed aggregates provided by a *view index*. N1QL, its query language, has many of the capabilities of SQL, but in general, its internal query processing architecture has very much been aimed at handling small, low-latency queries. This can be seen by comparing and contrasting the query execution pipeline described later in Section 4.5.3 with the data-parallel strategies found in parallel relational database systems that target large, warehouse-style use cases [5].

The longer-term goal and vision of Couchbase Server is to provide a single, unified platform that can be used for nearly all types of operational workloads – including operational analytics. Going forward, the support of such new analytic functionality cannot be added at the expense of the (sacred!) front-end OLTP workloads. To protect the front-end, there are some core principles that have been incorporated in the architecture and will be applied to future architectural changes as well.

2.3 Core Design Principles

2.3.1 Scaling workloads independently

This first core principle is applied to any service that manages data or data derived from the core key-value data set. Termed multi-dimensional scaling, this concept allows for services to be scaled up or out independent to one another from a hardware perspective. Each database workload has different needs, with some needing more I/O bandwidth and some needing more memory. Using this concept, hardware can be optimized based on the workload also making for more efficient use of hardware resources. This principle has been applied to Couchbase Server's data service, index service, and query service as well as to the services that will be introduced in the future like those for text search and analytics. Details of the implementation can be found in Section 4.4.

2.3.2 Asynchronous approach to everything

Relational databases mostly take a synchronous approach to data handling, which implies incurring a latency "hit" (penalty) at the time of a write (e.g., update, insert, or delete). However, given that the need for higher throughputs and lower latencies has only increased over time, Couchbase Server made a design choice to update all other components of the database asynchronously when a data update occurs. As a result, no immediate latency penalty is incurred at write time. This approach is used even for persisting updated data to disk.It is important to recognize that the rates at which new data can arrive and existing data can change may ultimately still be I/O limited, as, in the limit, the system must still be able to absorb the load offered by an application. However, asynchrony "buys time" for the system to handle spikes in the load; it also provides an opportunity for repeated updates to an object to be aggregated at the level of persistence.

Couchbase Server's asynchronous approach results in a world in which the data and its on-disk copy, the data and its replicas, and/or the data and its indices may become slightly out of sync. To make it easier for users to manage in such a world, Couchbase Server provides options for durability as well as consistency.

Durability guarantees

At write time, Couchbase provides client applications with the option to wait for replication and/or for persistence on a per mutation basis. This allows each application to make its own choice, based on its requirements, regarding taking (or not) the performance penalty associated with these options. Given that the latency expectations for modern large-scale OLTP applications are in sub-milliseconds, most users choose to receive a response immediately once the data hits memory or in some cases may choose to first replicate the data to one other node for safety. Since replication is memory-to-memory, the latency hit with the replication option is significantly less than waiting for persistence, especially when using spinning disks.

Configurable consistency for queries

Given that indexes in Couchbase are asynchronously updated, they can be out of sync with the data when a query makes a scan request. Couchbase supports two flavors of querying: N1QL and view queries. Both flavors allow users to specify the tolerable level of staleness on a per query basis. This means that users can choose between accessing an index as it is or waiting for the index to be updated up to the point in time when the query was initiated. The implementation of this concept for *view* queries is discussed in Section 3.1.2, and Section 3.2.3 discusses its implementation in the case of N1QL query processing.

2.3.3 Memory-first architecture

Couchbase follows a memory-first architecture when reading data as well as when distributing updates to the various components of the system. Almost all components are updated via DCP, which is the server's internal, in-memory database change protocol. This memory-first architecture makes it easier for all components to keep up with the system's high-throughput front-end loads.

3. COUCHBASE SERVER DATA BASICS

Couchbase Server is a distributed document database that supports various types of access paths and is built to scale in multiple ways [4]. It has a memory-first architecture that includes both an integrated caching layer and in-memory replication capabilities.

Couchbase Server stores data in JSON documents, where each document is a JSON object consisting of a number of fields. An application object can be stored using one or more documents, and the object's attributes become fields in the document(s). Documents are stored within a key space called a *Couchbase bucket,* and they can be directly accessed using a (user-provided) document ID much as one would use a primary key for lookups in an RDBMS.

The generally available version of Couchbase Server is currently version 4.1, with a preview of version 4.5 being available as of this writing. This paper's architectural description is thus based primarily based on version 4.1. The futures section will provide more insights into the system's upcoming capabilities and plans, including some of the highlights of version 4.5.

3.1 Client Access

Couchbase provides numerous client SDKs, including support for Java, .NET, php, Ruby, C, Python, node.JS and GoLang. Each of the client SDKs provides language-specific APIs or methods to access data via the various access paths available.

There are three main access paths by which a client application can talk to Couchbase Server:

1. Read / write JSON documents using key-value access via the primary key
2. Read / query JSON documents using the View API
3. Read / query JSON documents using N1QL queries

3.1.1 Key-value access using the primary key

Read access to DB

Couchbase provides a key-based lookup mechanism where the client is expected to provide the key, and only the cluster node hosting the data with that key will be contacted.

Write access to DB

Couchbase updates happen at the document level if the key-value API is used. A client SDK will retrieve a document that needs to be updated from the server, the user will modify certain fields, and the client SDK will then send the document back to the server for update.

To provide the required degree of isolation for concurrent access, Couchbase provides a CAS (compare and swap) mechanism for optimistic locking:

- When the client retrieves a document, a CAS ID (much like a revision number) is attached to it.
- While the client is manipulating the retrieved document locally, another client may modify this document. If this happens, the CAS ID of the document at the server will be incremented.
- Now, when the original client submits its modification to the server, it can choose to include the original CAS ID in its request. The server will then check this ID against the current ID in the server. If they differ, the document has been updated in between and the server will not apply the update.
- The original client can re-read the document (which now has a newer ID) and re-submit its modification.

Couchbase also offers its users a stricter locking mechanism; an application can opt to request a hard lock at the document level when performing its updates. (This lock will be released after a certain timeout to avoid deadlocks.)

Optimistic locking is what most large-scale online applications and production deployments use, as isolation is important to them, but not at the cost of a major performance penalty.

3.1.2 Read / query JSON using View API

Similar to the materialized view concept in the RDBMS world [11], Couchbase Server provides a MapReduce-style index called a *view.* A Couchbase *view* is essentially just a local (distributed) index that can be queried. A view is defined using a *Map function* that extracts data from the documents in a key space (bucket) and optionally a *Reduce* function that aggregates the data objects emitted by the map function. These are defined using JavaScript. A Couchbase view thereby provides a functional way to specify a materialized (and distributed) query result that client applications can directly utilize to improve their performance.

Queries that use views are static, meaning that a view needs to be defined and materialized before it can be used for querying data. An *emit( )* function that must be included in the *Map( )* function of a view is similar in its role to the query that would be specified in an RDBMS's CREATE MATERIALIZED VIEW statement. Every document that is a part of the key space upon which a view is defined will be processed by the view's *Map( )* and *Reduce( )* functions.

Once a view is materialized, it can then be queried for specific key(s) or a range of keys as follows:

- Return all values (as JSON) matching the supplied key, or
- Return all values (as JSON) matching any of the supplied keys, or
- Return all values (as JSON) starting with the provided key A and stopping on the last instance of a key B.

A given *view query* will be broadcast to all servers in the cluster and the results will be merged and sent back to the client SDK. To clarify all of these concepts, let us consider a simple example.

Sample document with key 'borkar123'

```
{
  ''name'': ''Dipti Borkar'',
  ''email'': ''Dipti@couchbase.com''
}
```

Example: Definition of View *Profile*

```
Function(doc){
if (doc.name){
       emit(doc.name, doc.email) }
```

This *map( )* function when executed will materialize a view and store the names and emails for all of the documents in the bucket that it is defined on. This *Profile View* can then be directly queried via the REST API or using a client SDK.

Example: REST query for View *Profile*

```
?key="Dipti"&stale=false
```

This REST call will return the value of the *doc.email* attribute from all documents where *doc.name* is “Dipti”.

Views are eventually consistent [9] with respect to the underlying stored documents; they are kept up-to-date asynchronously, on demand, based on document writes/updates. As explained earlier, Couchbase provides users a choice to run a view query with configurable consistency. A *stale* parameter can be used as a part of the view query to specify the required level of consistency (i.e., the tolerable degree of staleness) for the query.

Supported values for the *stale* parameter are:

- *false*: Wait for the view indexer to finish processing changes that correspond to the current key-value document set and then return the latest entries from the view index.
- *ok*: Just return the current entries from the index file (including possibly stale entries).
- *update_after*: Return the current entries from the index, but then initiate a view index update. (This is the default.)

3.1.3 Read / query JSON using N1QL

Couchbase Server now prominently supports query-based access to data using its new SQL-inspired query language, N1QL, as described next. N1QL queries can enter the system via a REST call, an SDK call, or one of several interactive client tools.

3.2 The N1QL query language

3.2.1 Overview and key capabilities

Non-first Normal Form Query Language, or N1QL (pronounced "nickel"), is the first NoSQL query language to leverage the flexibility of JSON with nearly the full expressive power of SQL and an SQL-friendly syntax. Developed by Couchbase for use in Couchbase Server, N1QL provides a common query language and a JSON-based data model for distributed, document-oriented database management. (A nice comparison of various NoSQL query languages and associated systems can be found in [6,13].)

N1QL enables clients to access data from Couchbase Server using SQL-like language constructs, as N1QL’s design was based on SQL. It includes a familiar data definition language (DDL), data manipulation language (DML), and query language statements, but can operate in the face of NoSQL database features such as key-value storage, multi-valued attributes, and nested objects.

3.2.2 Features of N1QL

N1QL provides a rich set of features that let users retrieve, manipulate, transform, and create JSON document data. Its key features include:

`SELECT` Statement: The `SELECT` statement in `N1QL` extends the functionality of the SQL `SELECT` statement to work with JSON documents. Of particular importance are the `USE KEYS, NEST`, and `UNNEST` sub-clauses of the `FROM` clause in N1QL.

At a high level, a `SELECT` statement supports the retrieval of data from specified keyspaces or Couchbase buckets. A simple query in N1QL has three parts to it:

`SELECT` - Portions of the document to return.

`FROM` - The keyspace or datastore with which to work.

`WHERE` – Conditions the retrieved data should satisfy.

Because data in Couchbase Server is stored in documents rather than in rigidly structured tables, N1QL queries can actually return a collection of different document structures or fragments.

Data Manipulation Language (DML): N1QL provides support for `INSERT`, `DELETE`, `UPDATE`, and `UPSERT` statements to create, delete, and modify data stored as JSON documents. These statements also support sub-document level lookups and updates.

3.2.3 Salient clauses and parameters

While N1QL supports an extensive set of clauses similar to SQL, there are certain clauses and parameters that differentiate it from other databases. Let us examine some of the key SQL clause additions that N1QL includes:

USE KEYS

This clause is the key (to so speak) to bridging the fundamental functionality gap between a key-value store and a document database. Its use provides the user with the flexibility to grab specific attributes or compose new values from existing objects while still getting key-value retrieval performance.

Specific primary keys within a key space (*Couchbase bucket*) can be specified in this clause. Only values having those primary keys will be included as inputs to the rest of the given N1QL query.

Example: To specify a single key:

```
SELECT * FROM profiles USE KEYS "acme-uuid-
1234-5678"
```

Example: To specify multiple keys:

```
SELECT * FROM profiles
USE KEYS ["acme-uuid-1234-5678", "roadster-
uuid-4321-8765"]
```

NEST and UNNEST Clauses

Supporting the JSON data format requires Couchbase to provide extensions to standard SQL to allow users to do more with JSON. Specifically, as JSON objects can be stored in the database as-is, without splitting or shredding them into multiple tables, users may need to flatten documents when reading them out, or if related documents are stored separately, to compose and combine them together into a single JSON document. Both of these goals can be achieved using the N1QL `UNNEST` and `NEST` clauses.

When the `NEST` clause is used, instead of producing a cross-product of the left and right hand inputs, it produces a single result for each left-hand input while its right-hand input is collected into an array and nested into a single array-valued field in the result.

Example: Consider a bucket that contains two types of documents, with *doc_type* ‘user_profile’ and ‘order’. If we wish to assemble a list of products purchased by a user, we need to perform a `JOIN` across those two document types. However, a traditional relational join would yield a cross product instead of composing a result within which, for each user profile that matches the query criteria, its associated orders are embedded or nested within an array.

```
SELECT PO.personal_details, orders
FROM profiles_orders PO
USE KEYS 'borkar123'
NEST profiles_orders as orders
    ON KEYS ARRAY s.order_id FOR s
    IN PO.shipped_order_history END
```

The query above will return a JSON document with all the orders placed by the user whose key is 'borkar123' being nested into an array within the resulting user object.

In contrast, the UNNEST clause takes the contents of a nested array and joins them each with their parent object.

Example: Return a list of the existing (in-use) product categories in a world where products can fall into multiple categories

```
SELECT DISTINCT (categories)
FROM product
UNNEST product.categories AS categories
```

Query Scan Consistency

The consistency level for a given N1QL query can be configured using the staleness parameter for the query. The following consistency levels can be specified for the staleness parameter:

- `scan_consistency=not_bounded`

This level returns the query with the lowest latency, as it is the most relaxed consistency level. Selecting this option essentially means the query can return data that is currently indexed and accessible by the index or the view. The query output can be arbitrarily out-of-date if there are many pending mutations that have not been indexed by the index or the view. This consistency level is useful for queries that favor low latency and do not need the most up-to-date information.

- `scan_consistency=request_plus`

This level provides the strictest consistency level and thus executes with higher latencies than the other levels. This consistency level requires all mutations, up to the moment of the query request, to be processed before query execution can begin. This ensures that any writes that are done prior to issuing the query request, and possibly more recent mutations, have been indexed by the GSI or the view indexer and will be returned by the N1QL query if it qualifies for the result set. This guarantee is important to applications that require consistent reads or read-your-own-write semantics.

3.2.4 Restrictions in N1QL

While N1QL is very powerful (i.e., general), including its SQL extensions for dealing with nested objects in JSON, there are certain SQL language features supported by relational database management systems that – if permitted in Couchbase Server – could be very detrimental to runtime performance, given the larger data sets and the unnormalized approach to managing data in the NoSQL world. One example of this would be expensive (general, non-key) joins, as opposed to those joins needed to quickly traverse key-based relationships. A restricted Cartesian product across two secondary attributes of documents is not supported linguistically in N1QL. Instead, joins are only allowed when one of the two sides involves the primary key (document ID) within a bucket.

3.3 Indexing support

As would be expected of any database system, Couchbase Server provides indexing capabilities to speed up query performance. It provides two types of indexes.

3.3.1 Local View Index

As explained earlier, one type of index provided is the *view* index. Views are built asynchronously and are maintained for each mutation. Couchbase Server pre-computes and stores these view results before returning results to a client. Views are local indexes, i.e., they are co-located on each node where their associated indexed data lives, so they use the same hash-partitioning scheme discussed later for the data (see Figure 5).

Views can either be defined using Javascript (as in the earlier example) or via a SQL-like `CREATE INDEX` statement.

Example: Create an index using a local view on the email attribute of the profiles stored in the 'Profile' Couchbase Bucket

```
CREATE INDEX email on `Profile` (email)
USING VIEW;
```

3.3.2 Global Secondary Index

As their name suggests, a global secondary index (GSI) is a global index on all of the documents stored within a specified Couchbase bucket, and it is stored separately (hence "global") from the data itself. It can optionally use a range-partitioned scheme separate from the data's hash-partitioning scheme. GSI indexes reside on the nodes of the cluster where the index service is running. (Figure 9 in Section 4.3 shows a sample cluster topology.)

Example: Create an index using a GSI index on the email attribute of the profiles stored in the 'Profile' Couchbase Bucket

```
CREATE INDEX email on `Profile` (email)
USING GSI;
```

3.3.3 Primary Index

Typically, when describing the indexing capabilities of a database, one would expect to have the Primary Index described first. However, in the case of Couchbase Server, the "real" primary index in Couchbase is the hash table of keys that allows users to perform direct key value access. However, since this table doesn't provide a mechanism to perform range queries and other types of lookups on the attributes of JSON, Couchbase additionally allows users to define a PRIMARY INDEX similar to that of a RDBMS but based on the server's secondary indexing capabilities (either using a local view or a global secondary index) described above.

Example: Create a Primary Index on the Profile bucket

```
CREATE PRIMARY INDEX profile_pk_view ON
Profile USING VIEW;
```

or

```
CREATE PRIMARY INDEX profile_pk_gsi ON
Profile USING GSI WITH
{''defer_build'':true''};
```

3.3.4 Selective or Partial Indexes

Given that the data is stored in Couchbase is in a schema-free and unnormalized format, and that in many cases, documents (objects) of different types may be stored together in a given Couchbase bucket, N1QL allows users to create "selective indexes" (also sometimes called partial indexes [8]). Filtering the data to be indexed based on the anticipated query workload can significantly improve both indexing and query performance.

Example: Create an index using a global index on the age attribute of the profiles stored in the 'Profile' Couchbase bucket, but just for data where the age is > 21

```
CREATE INDEX over21 ON `Profile`(age)
WHERE age > 21
USING GSI;
```

4. A LOOK UNDER THE HOOD

4.1 Overview of clustered architecture

Couchbase Server has a shared-nothing architecture. It is typically set up as a cluster of multiple servers behind an application server. A cluster of Couchbase Servers consists of one or more nodes, with each containing a configurable set of services. This architecture allows the cluster to be scaled out horizontally, by adding more nodes, or scaled up vertically, by adding "beefier" hardware.

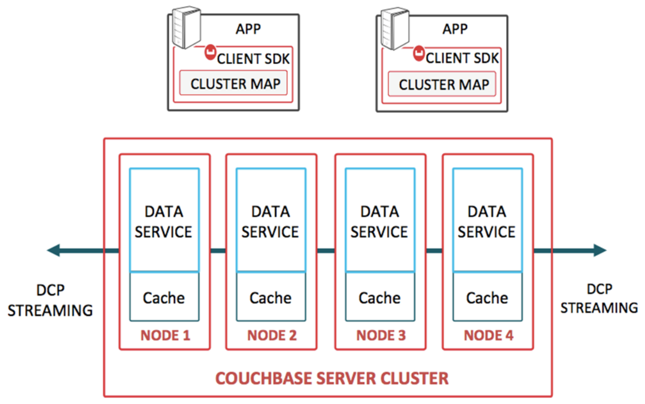

Figure 4. Couchbase Cluster Architecture

Figure 4 depicts a 4-node cluster. The figure's two application servers run the Couchbase client driver, and these clients can talk with any node in the cluster to read and/or write data. In this diagram, only the data service is running. There are additional services (described later) that can be added as the functionality required for an application becomes more complex.

A Couchbase bucket is the rough equivalent of a database and is better described as being a key space. Documents with different schemas can be stored in the same bucket. Each bucket is split into 1024 logical partitions called *vBuckets (vB)*. This is not a configurable number; any Couchbase server deployment has 1024 logical partitions. This number was selected to provide a good balance of cluster manageability from a size perspective and the amount of data that is managed per partition. The former is important for operational maintenance, and the latter is important for the performance of maintenance tasks that are carried out across the cluster. In general, cluster sizes as well as the amount of data managed by them have grown in production deployments; however, given the operational focus, in practice the number of nodes in our production clusters has yet to grow beyond about 150 nodes. A 150-node cluster can support many terabytes of data and has the capacity to handle millions of operations per second. Section 10.1.1 presents an overview of the Key Value performance.

vBuckets are mapped to physical servers across the cluster, and the mapping is stored in a lookup structure called the **cluster map**. Section 4.3.1 goes into more details about how this map is managed. Applications can use Couchbase's smart clients, which contain a copy of the cluster map, to interact with the server. Smart clients can hash the **document ID (key)**, which is specified by the application, for each document added to Couchbase. As shown in Figure 5, a client applies a hash function (CRC32) to every document that needs to be stored in Couchbase, and the document can then be sent directly from the client to the server where it should reside.

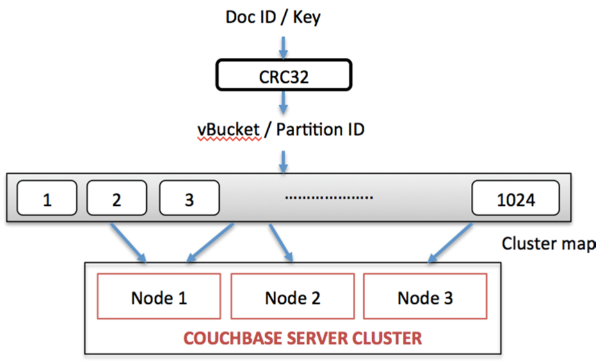

Figure 5. Mapping Keys to Partitions and Servers

4.1.1 Intra-cluster Replication

A bucket can be replicated up to 3 times, giving the user up to 4 copies of their data. An individual server manages only a subset of the active and replica partitions. At any point in time, for a given partition, only one copy of the partition will be active, with zero or more replica partitions on other servers. If the server hosting an active partition fails, the cluster will promote one of the replica partitions to active status, thereby ensuring that applications can continue to access the data without incurring downtime.

4.2 Asynchronous architecture

As explained earlier, in Couchbase Server all of the operations after a document is first written to memory will be performed *asynchronously*. This is a fundamental concept that extends to every facet of the server.

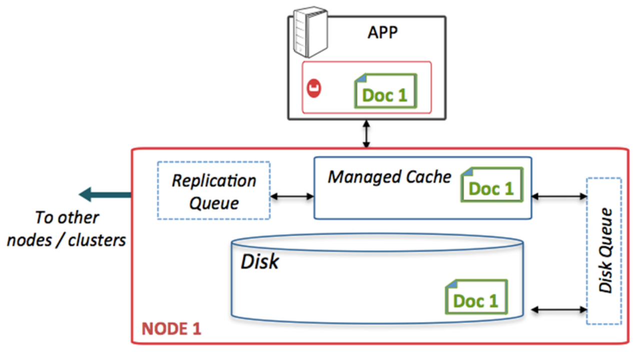

Figure 6. Couchbase Server Asynchronous Architecture

When data is written to Couchbase, it is first stored in the hash tables in the integrated (managed) cache. At this point, an initial acknowledgement of receipt of the mutation is sent back to the client SDK. This mutation is then asynchronously written to disk via the disk write queue, and at the same time it is also pushed into the in-memory replication queue to be replicated to other nodes within the cluster. This is summarized in Figure 6.

As explained in the earlier discussion of durability constraints, client applications are given a choice of whether or not to wait for replication and/or for persistence on a per mutation basis. This allows each application to make its own choice, based on its requirements and the nature of its data, regarding taking (or not) the performance penalty associated with these operations. When a document it written, a sequence number is generated and associated with the mutation. The maximum sequence number per vBucket is also tracked. This mechanism allows other parts of the system to provide an option for strong consistency.

Each mutation is also asynchronously fed to other parts of the system, such as cross-cluster replication, the view index engine, and the global secondary index (GSI) service. If a N1QL query chooses `request_plus` scan consistency, the query engine will wait until the index is updated up to the maximum sequence number for each vBucket.

4.3 Node-level architecture

The nodes in a Couchbase Server cluster can all look the same, or various subsets of the cluster nodes can be configured to run a particular (sub)set of services. This uniformity of architecture allows users to seamlessly scale a database linearly without a single point of failure. However, independent of what other services are running, there is some core functionality that runs on every node, *e.g.,* the cluster manager.

4.3.1 Cluster Manager

The Couchbase cluster manager supervises server configuration and interaction across all servers within a cluster. It is a critical component, as it manages the replication and data rebalancing operations in Couchbase. Although the cluster manager executes locally on each cluster node, the nodes also elect a cluster-wide *orchestrator node* to watch over the cluster conditions and carry out appropriate cluster management functions.

If a node in the cluster crashes or otherwise becomes unavailable, the orchestrator notifies all other machines in the cluster. It promotes to active status replica partitions associated with the server that went down. The cluster map will also be updated on all of the cluster nodes and the clients. The process of activating replica partitions is known as *failover***.**

If the orchestrator node itself crashes, the existing nodes will detect the fact that it is no longer available and they will elect a new orchestrator immediately so that the cluster can continue to operate without disruption.

When the number of servers in a cluster changes due to scaling out or due to failures, data partitions must be redistributed. Doing this ensures that data will remain evenly distributed throughout the cluster and thus that application access to the data will be load-balanced evenly across all the servers. This process is called **rebalancing.**

When a rebalance begins, a new cluster map is calculated based on the current pending set of servers to be added and removed from the cluster. It is then streamed to all the servers in the cluster. During rebalance, the cluster moves the data directly between two server nodes (the source and destination) in the cluster. Once the cluster moves each partition from one location to another, an atomic and consistent switchover takes place between the two affected nodes, and the cluster updates each connected client library with the new cluster map. Throughout the migration and redistribution of partitions among servers, any given partition on a server will be in one of the following states:

Active: The server hosting the partition is servicing all types of requests for this partition.

Replica: The server hosting the partition cannot handle client requests, but it will receive replication commands. Rebalance marks the destination partitions as being replicas until they are ready to be switched to active.

Dead: This server is not in any way responsible for this partition.

4.3.2 Database Change Protocol

Any mutation that happens on an object in the data service must be propagated to all other parts on the system that need to know, including data replication, indexes, and so on. Couchbase has an internal Database Change Protocol (DCP) that is utilized to keep all of the different components in sync and to move data between the components at high speed. DCP lies at the heart of Couchbase Server and supports its memory-first architecture by decoupling potential I/O bottlenecks from many critical functions.

4.3.3 Data Service

As described above, Couchbase Server supports high speed, direct read/write operations based on a core Key-Value API (KV API). The Data Service provides the KV API that allows developers to create, retrieve, update and delete records by primary key. The Data Service forms the base data management layer of Couchbase and is leveraged by the Indexing and Query services. The data manager has a number of parts, including the object-managed cache, storage engine, view engine, and projector and router.

Object Managed Cache

Key-value pairs are stored in the object-managed cache. Hash tables for each virtual bucket reside in this cache and offer a quick way of detecting whether a given document currently exists in memory or not. Each document in a partition will have a corresponding entry in the hash table; each entry for a document stores the document's ID (i.e., its key), some document metadata, and the document's value.

By default the key and the metadata for every key in the bucket will be kept in memory, while the associated values can be evicted based on usage. Users also have the option to enable the eviction of the key and metadata based on usage.

Storage Engine

With Couchbase's append-only storage engine design, document mutations always go to the end of a file. When a new document is added in Couchbase, it is added at the end of a file. If an update is made to a document in Couchbase, it also is recorded at the end of the file. This improves disk write performance, as all updates are written sequentially. *Compaction* is periodically run, based on a fragmentation threshold, and while the system is online, to clean up stale data from the append-only storage.

View Engine

The view engine is responsible for building and querying ***view indexes.*** As explained earlier, views in Couchbase are defined using a map function, which extracts the data of interest from documents, and an optional reduce function, which aggregates the data that is emitted by the map function. Since the view index is a local index, the view engine runs within the data service. This component is a consumer of the DCP feed of the mutations needed to update the view indexes.

During initial view building or materialization of the view, Couchbase reads the partition's data files and applies the map function across every document. A key characteristic of a view index is that it stores the pre-computed aggregates defined in the Reduce function as a part of the index tree. This allows for very fast aggregation at query time.

As shown in Figure 8, applications can query views using one of the Couchbase SDKs. View query execution takes a scatter/gather approach: Queries are sent to a randomly selected server within the cluster. The server that receives a query sends the request to the other relevant servers in the cluster and then aggregates their results. The query's final result is then returned to the client.

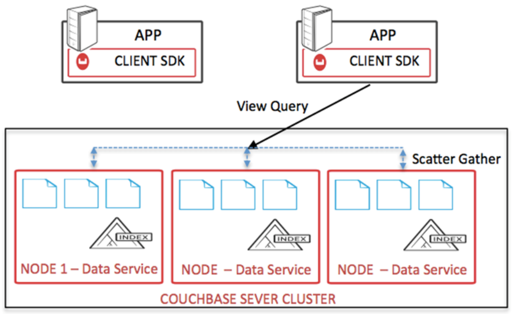

Figure 8. Distributed View indexing and querying

During a rebalance operation, partitions change state – the original partition transitions from *active* to *dead*, and the target partition transitions from *replica* to *active*. Since the view index on each node is synchronized with the partitions on the same server, when a partition has migrated to a different server, the documents that belong to the migrated partition should not be used in the view result anymore. To keep track of the relevant partitions that can be used in the view index, information about vBuckets is stored in the view B-tree itself. Using this information, parts of a B-tree can be deactivated as needed. This helps maintain consistency when querying a view index during rebalancing or failover operations.

Index Projector

The Projector is responsible for mapping incoming mutations to a set of Global Secondary Key Versions needed for secondary index maintenance. The Projector resides within the data service where the mutation originated, and it is a consumer of the DCP feed of mutations that sends its evaluated results to the Router.

Index Router

The Router is responsible for sending Key Versions to the index service. The router relies on the index distribution and partitioning topology to determine which indexer(s) should receive the key version. The router resides on the same node as the projector.

4.3.4 Index Service

The indexing service is responsible for managing all of the global secondary indexes. Various components work together to manage and maintain global secondary indexes. The projector and router live on the data service, while the index managers and indexers live on the index service. These together cooperate to manage the mutations coming from the Data Service via DCP. The projector extracts the secondary keys relevant to the indexes that have been defined and sends them to the router. The router then decides which indexer to send the message to. In case the index is partitioned, the partition key tells the router which indexer and which node to send the message to. An insert message may be sent to one indexer with a delete message being sent to another in the event that the value of the partition key itself has changed.

Index Manager

The *Index Manager* resides within the indexing service (see Figure 9) and is responsible for receiving requests for indexing operations (*e.g.,* creation, deletion, maintenance, scan, lookup).

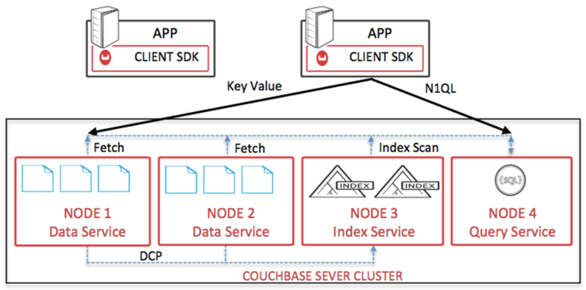

Figure 9. Index Service with Global Index

Indexer (Local Indexer)

The indexer component processes the changes received from the router and manages the on-disk index tree data structure. It also provides the interface for the query client to run index scans and it does scatter/gather for queries in case of a partitioned GSI index.

4.3.5 Query Service

At a high level, the Query Service takes an application query and performs the necessary functions to retrieve, filter, and/or project the data in order to resolve the application's request. Query execution is the job of the Query Service. To process a given user query, the query engine will issue requests to the index service, the data service, or both, depending on the chosen query plan.

Query Catalog

This Query Service component provides catalog support for the Query Service. This includes allowing it to perform Index DDL (*e.g.,* Create, Drop) and Index Scan/Stats operations. (Section 4.5 will discuss more of the details of query planning and execution.)

4.4 Multi-dimensional Scaling

As the earlier Sections described, an administrator can choose to run the Data, Index and Query Services on all or different nodes. This ability to have multiple "dimensions" in which to scale the cluster is called multi-dimensional scaling (MDS). This allows Couchbase users to scale workloads independently based on their needs – so a cluster can reflect and support the technical business requirements of an application.By separating its core data management functions into these three services, Couchbase Server enables users to control the scalability characteristics of their system based on workload characteristics. As those requirements change, Couchbase users can expand or shrink their cluster, adding data management resources to the function(s) where they are needed most in order to address the changing Data, Index, and Query Service requirements. A typical setup might have the Data Service running on nodes with more memory, the Query Service on nodes with more cores, and the Index Service on nodes with faster disks.

4.5 Deeper dive into indexing and querying

4.5.1 High-Level Query Service Interactions

Couchbase SDKs can route N1QL queries to any one of the nodes running the query service. The receiving node will analyze the query, use metadata on its referenced objects to choose the best execution plan, and execute the chosen plan. During execution, depending on the query and the available indexes, the query node works with the index and data nodes to retrieve keys and data, performing the required set of select-join-project operations.

The index service does not directly communicate with the data service. Instead, the query service issues all key-value access requests (unless a covering index can fully answer the query). An index simply returns the document ID for each attribute match found during index scans. This ID is then used by the query service to fetch the document itself in order to access and project out the additional fields that are needed.

4.5.2 Step-by-Step Query Execution Flow

Figure 10 enumerates the query processing steps for an example N1QL query. Once a query plan has been constructed, based on the operators and scans needed, the query service coordinates first with the index service and then with the data service. The query results are streamed to the client as they become available.

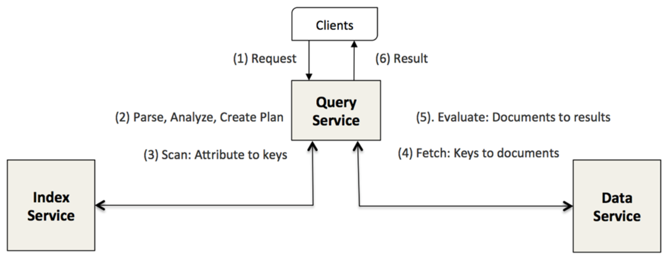
Clients
(1) Request
(6) Result
(2) Parse, Analyze, Create Plan
Query Service
(5). Evaluate: Documents to results
(3) Scan: Attribute to keys
(4) Fetch: Keys to documents
Index Service
Data Service

Figure 10. Data flow from query to index and data service

4.5.3 Query Execution Operators

Figure 11 shows the operators of query execution in more detail. Note that not all queries will have every operator in their plan, and some operators may appear and be executed multiple times. For example, the sort operator can be omitted if there is no ORDER BY clause in the query, while the scan, fetch, and join operators may be needed multiple times to perform multiple joins.

To see how a given N1QL query will be executed, an `EXPLAIN` statement can be used before any N1QL statement to request information about the execution plan for the statement.

Example:

```
EXPLAIN SELECT title, genre, runtime
FROM catalog.details
ORDER BY title
```

To optimize a query, the N1QL query planner analyzes the query and available access path options for each keyspace (bucket) in the query to pick an appropriate plan and execution infrastructure. The planner needs to first select the access path for each bucket, determine the join order, and then determine the type of the join operation. Once these big decisions have been made, the planner then creates the infrastructure needed to execute the plan. Some operations, like query parsing and planning, are done serially, while other operations, like fetch, join, and sort, are done in a local parallel (based on multicore) manner.

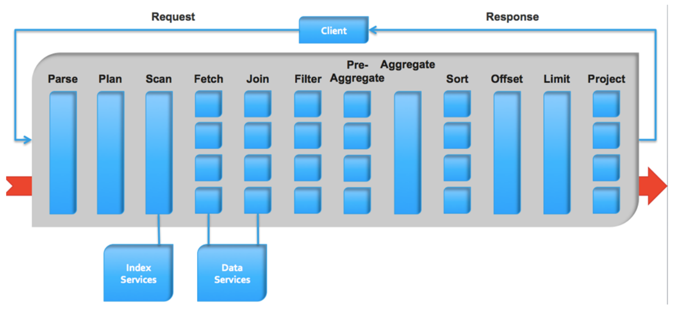

Figure 11. Operators in a N1QL query plan

In terms of the infrastructure available for query execution, some of the most important query plan operators are explained below:

Keyspace (bucket) scan – There are three types of scans:

- *Keyscan access:* When specific document IDs (primary keys) are available, the Keyscan access method retrieves the documents for those IDs. Any filters associated with that keyspace will be applied after retrieval. The Keyscan access method can be used either when a keyspace is being queried by itself or during join processing. A keyscan is commonly used to retrieve qualifying documents from the inner keyspace during join processing.

- *PrimaryScan access:* This is the equivalent of a full table scan in a relational database system. This is chosen when Documents IDs are not available and no qualifying secondary access methods are available for this keyspace. N1QL will again apply applicable filters on each document after retrieval. This access method is quite expensive, and the average time to return results increases linearly with number of documents in the bucket.

- *IndexScan access:* A qualifying secondary index scan is used to first filter the keyspace and determine the qualifying document IDs. The query executor then retrieves the qualifying documents from the data store by ID. In Couchbase, the chosen secondary index can be a (local) view index or a global secondary index.

Fetch – This operator reaches into the data service for the objects with a specific key. An index only contains document IDs, so the fetch operator is needed whenever a query includes additional projections that cannot be answered from the index alone. The execution of the fetch operator is parallelized.

Projection – There are two kinds of projection operators:
- *InitialProject*: This operator reduces the stream size to just the fields actually involved in query.
- *FinalProject*: This operator performs the final shaping of the result into the requested JSON schema.

Unnest – This is a join operation between a parent and a child object containing a nested array. In its result, the parent object is repeated for each child array item.

Nest: This inverse of **Unnest** is a grouping operation between a parent and a (desired) child with a nested array. In the result, the child array will be embedded in the parent object.

Join methods

N1QL supports a nested loop-based access method for all of the join types that it supports: `INNER JOIN` and `LEFT OUTER JOIN`. Consider the following `FROM` clause in a N1QL query:

```
FROM (ORDERS o INNER JOIN CUSTOMER c
ON KEYS o.O_C_ID)
```

For this join, ORDERS will serve as the outer keyspace and CUSTOMER will be the inner keyspace. The ORDERS keyspace will be scanned first (using one of the scan options described above). Then, for each of the qualifying documents from ORDERS, a KEYSCAN will occur on CUSTOMER based on the key O_C_ID in the ORDERS document.

4.6 Cross Cluster Replication

Cross datacenter replication (XDCR) provides a way to replicate active data to multiple, geographically diverse datacenters. This is done either for disaster recovery or to bring data closer to users.

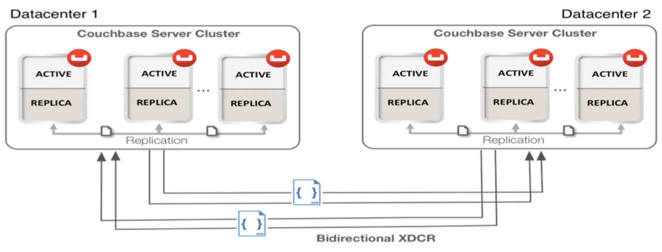

Figure 12. Cross datacenter replication architecture

Both intra-cluster replication and XDCR occur simultaneously, as Figure 12 illustrates. Intra-cluster replication takes place on both datacenters, and it happens within each cluster. At the same time, XDCR serves to replicate documents across datacenters. XDCR is also a consumer of the internal DCP stream, as it uses the DCP stream to push in-memory document mutations to the destination cluster. Some of the most notable attributes of XCDR are as follows:

XDCR can be setup on a per bucket basis. Depending upon application's requirements, only a subset of data may need to be replicated. This can be done either on a per bucket basis or even within a bucket by using filtered replication (based on a regular expression on the document ID, i.e., primary key, string).

XDCR is **cluster topology aware**. The source and destination clusters can have different numbers of servers and thus different data partitioning. If a server in the destination cluster goes down, XDCR is able to utilize the updated cluster topology information and can continue replicating data to the appropriate available servers in the destination cluster.

4.6.1 Consistency Across Clusters

Within a cluster, Couchbase Server provides strong consistency at the document level for the key-value API and controllable consistency when querying. Across clusters, XDCR provides **eventual consistency** [9]. This makes Couchbase a CP system within a cluster but an AP system across clusters with respect to the CAP theorem [12]. This choice was made to avoid taking a performance hit on writes, maintaining high performance while still providing 100% availability for reads / writes across clusters.

A built-in conflict resolution mechanism picks the same "winner" on both clusters if a document is mutated on both before being replicated. If such a conflict occurs, the document with the most updates is considered the "winner." If both clusters have the same number of updates for a document, additional metadata fields are used to pick the "winner." XDCR applies the same rule on both clusters to make sure that document consistency is maintained.

5. TUNING FOR OLTP PERFORMANCE

There are certain things commonly done in relational systems that would not work well with today's NoSQL data volumes and the scale of the systems needed to support them.

5.1.1 Avoid using a PRIMARY index

Couchbase allows users to create a PRIMARY index as a way to access all objects in a keyspace, but a best practice is to avoid queries requiring a PRIMARY index scan. The fastest data access will be via key-value look-ups or N1QL's USE KEYS clause.

5.1.2 Use of covering indexes

For the best performance, appropriate GSI indexes can be created to support a query without any document fetches. Such a covering index includes all of the information needed to satisfy the query and can thus avoid the need for an additional step to access the indexed data. Covered queries, that is, queries that get all their information from the index, deliver better performance. (This is actually a trick used in the relational world as well; it essentially uses an index as a "poor man's materialized view".)

6. A GLIMPSE INTO THE FUTURE

As described in the Introduction, Couchbase Server has evolved significantly since the early days, and its evolution is ongoing.

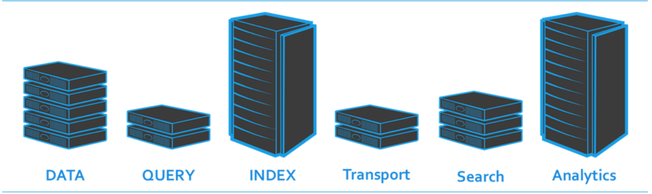

Figure 13. Services in a future Couchbase Server

With a vision to provide a single platform that can be used for all types of operational workloads, there are several key components that will be added to Couchbase in the near future. The resulting set of component services is sketched in Figure 13.

6.1 Near-term plans

6.1.1 Memory-optimized indexes

With a focus on its low-latency-driven memory-first architecture, a new feature in Couchbase version 4.5 is support for memory-optimized indexes.

These new indexes will reside completely in memory, dramatically reducing dependence on disk. Recoverability is provided via disk-backups. This functionality will allow users with very high write-heavy workloads to continue to utilize N1QL and indexing in addition to key-value access, as indexes can keep up with higher mutation rates. It will also allow very fast index scans to reduce query latencies. Section 10.1.2 presents an overview of query performance with memory-optimized indexes.

6.1.2 *Array indexes*

One of the key strengths of JSON is its ability to nest objects or items within arrays. Arrays are one very natural approach used to embed related data objects within a parent (e.g., think line items within orders). Nesting provides great flexibility from a data modeling perspective, but it is also important for those individual objects from within an array to be indexed and queried. N1QL today allows users to query items within an array, but without an index to support array predicates, the performance of such queries can be an issue. Version 4.5 will enable users to create indexes on array-valued fields, making those queries significantly faster.

6.1.3 *Full-text search service*

Another workload dimension that is required for some operational applications is full-text search. This is typically based on a reverse index, where all the *words* within the data are indexed to be able to do term-based, phrase-based, and/or prefix-based searches. Full-text search is another type of service currently being added that will receive data mutations via in-memory DCP and will be able to be scaled up or out independently as well.

6.2 Medium-term plans

One big area that Couchbase plans to add to the mix of services available to operational applications is operational analytics. By consolidating this into the core platform, rich applications can be built with a single platform that enables end-to-end management of operational data. Query-based insights from the latest updates can then be fed back into applications almost instantly without having to move data physically out of the cluster or being forced to restructure (*e.g.,* flatten) it via pre-analysis ETL.

The planned analytics service will be based on the open-source Apache AsterixDB Big Data Management System [1, 2]. The NSF-sponsored Asterix project set out to develop a next-generation system to ingest, manage, index, query, and analyze mass quantities of semi-structured data. The resulting platform, AsterixDB, has an extremely complementary data model to that of Couchbase. That commonality is one of the main reasons that the two teams decided to collaborate. The planned analytical service will be another new service that is fed via in-memory DCP and that can be scaled either out or up independently with respect to other services, especially the data service (to provide performance isolation for the all-important front-end OLTP workloads). The new analytics service will support a much wider range of queries and will include a parallel database inspired scalable runtime engine. Under the hood, AsterixDB includes partitioned, LSM-based data storage and indexing, support for both natively stored and external data, and a rich set of built-in types. A typical workload on the analytics engine, unlike the OLTP-like N1QL query engine, will include richer (and more expensive) queries such as large joins, aggregations, grouping, and so on.

7. CONCLUSION

We have described the history and evolution of Couchbase Server from its early roots as a key-value store to its current form, which is a full-featured and fully queryable document data manager. We pointed out a number of the unique features and requirements that arose from the need to support low-latency, OLTP-like workloads. We described the view from outside the system and the APIs available to users. These external interfaces were designed to be simple, but the internal architecture that supports them was built to support the OLTP requirements of next-generation applications, including flexible schemas, incremental scalability, and high performance. We explained how the components of Couchbase Server work together under the hood, including its data storage, indexing, and query services, and we closed with a glimpse into where things are expected to be going in the future.

8. ACKNOWLEDGEMENTS

The existence of Couchbase Server 4.1 (and 4.5) today is due to the combined efforts of many terrific Couchbase team members whose contributions we wish to acknowledge.

9. REFERENCES

[1] S. Alsubaiee *et al,* AsterixDB: A Scalable, Open Source BDMS, *Proc. VLDB Endowment* 7(14), October 2014.

[2] *Apache AsterixDB*, http://asterixdb.apache.org/.

[3] R. Cattell, Scalable SQL and NoSQL Data Stores, *ACM SIGMOD Record* 39(4), May 2011.

[4] *Couchbase NoSQL Database*, http://www.couchbase.com/.

[5] D. DeWitt and J. Gray, Parallel Database Systems: The Future of High Performance Database Systems, *CACM* 35(6), June 1992.

[6] Y. Papakonstantinou, Semistructured Models, Queries and Algebras in the Big Data Era, Tutorial in *Proc. ACM SIGMOD Conf.*, San Francisco, CA, June 2016.

[7] P. Sadalage and M. Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*, Addison-Wesley Professional, 2012.

[8] M. Stonebraker. The Case for Partial Indexes, *ACM SIGMOD Record* 18(4), December 1989.

[9] D. Terry, Replicated Data Consistency Explained Through Baseball, *CACM* 56(12), December 2013.

[10] *Magic Quadrant for Operational Database Management Systems,* Gartner, https://www.gartner.com/doc/3147919/magic-quadrant-operational-database-management.

[11] R. Bello *et al*, Materialized Views in Oracle, *Proc. 24th VLDB Conf.,* New York, NY, August 1998.

[12] S. Gilbert and N. Lynch, Perspectives on the CAP Theorem, *Computer* 45(2), February 2012

[13] K. Ong, Y. Papakonstantinou, and R. Vernoux, *The SQL++ Query Language: Configurable, Unifying and Semi-structured*, http://arxiv.org/abs/1405.3631.

[14] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, Benchmarking Cloud Serving Systems with YCSB, *Proc. 1st ACM Symp. on Cloud Computing*, Indianapolis, Indiana, June 2010.

10. APPENDIX

10.1 Couchbase Server performance

In this section, we present some sample performance results, obtained using the Couchbase Server 4.5 Developer Preview, to provide a brief overview of the performance characteristics of Couchbase for both key-value and query workloads. The testing tool used was the Yahoo Cloud Serving Benchmark (YCSB) [14]. The Couchbase adapter for YCSB was built to operate against a Couchbase Server cluster using the official Couchbase Java SDK and provides a rich set of configuration options, including support for the N1QL query language.

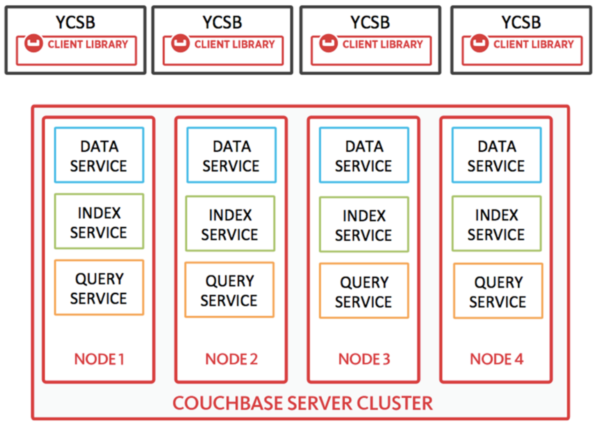

Figure 14. Performance test setup

The deployment topology used was simple, with the data, index and query services running on all nodes of a 4-node cluster. There are 4 YCSB clients that generate load. The number for threads for each client was scaled up from 12 threads to 32 threads and the maximum throughput was measured. There are two workloads presented here; each was run on the 4-node Couchbase cluster using an early version (Developer Preview) of the 4.5 release.

10.1.1 *Key-value performance (YCSB workload A)*

Workload A of YCSB is a mixed workload with 50% reads and 50% writes of keys. In this test, the thread counts for each of the four YCSB clients were varied from 12 to 32 threads. A data set of 10 million documents was used. The throughput in operations per second was measured and is plotted in Figure 15. As shown, a 4-node cluster with a total of 128 client threads driving load is capable of running approximately 178K operations per second.

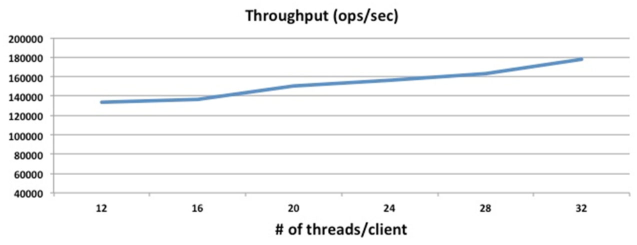

Figure 15. Simple operation throughput (ops/sec) *vs.* threads

10.1.2 *Query performance (YCSB workload E)*

Workload E of YCSB is a query workload consisting of small range queries. Short ranges of documents are queried via N1QL instead of performing individual document operations.

Sample query: `SELECT meta().id AS id FROM 'bucket' WHERE meta().id >= '$1' LIMIT $2;`

Similar to workload A, the thread counts for each of the four YCSB clients was varied from 12 to 32 threads. The throughput in queries per second was measured and is plotted in Figure 16 below. As shown, a 4-node cluster with a total of 128 client threads driving load is capable of running approximately 5400 small range queries per second.

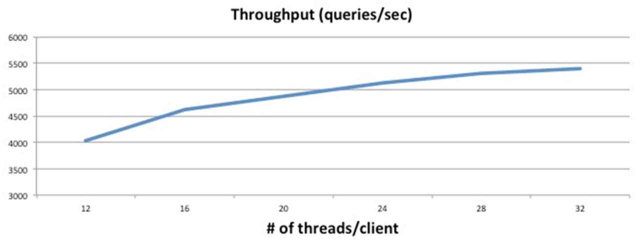

Figure 16. Range query throughput (queries/sec) *vs.* threads