

# Couchbase Server 4.5: What's New

From the team that brought you integrated caching, cross data center replication, N1QL, and surf and turf comes Couchbase Server 4.5 — because the only thing better than steak and lobster on the same plate is JSON and SQL in the same database.

However, it's not enough to combine a SQL-based query language with a flexible data model.

We also wanted to make the transition from relational to NoSQL even simpler and easier for you. How? By adding a web-based query editor and schema browser, and providing comprehensive query monitoring. While we were at it, we also added more security features and improved disaster recovery, too.

But we didn't forget about performance either. It's an obsession, and yes, we pushed the envelope in query performance. With N1QL, we gave you the most powerful query language in the land of NoSQL, but we also wanted to give you the fastest — and we did. The fact is, performance matters. In a world where success depends on customer experience and efficiency is critical, every millisecond counts. You need to get more out of your hardware, and with a faster, more efficient database, you can.

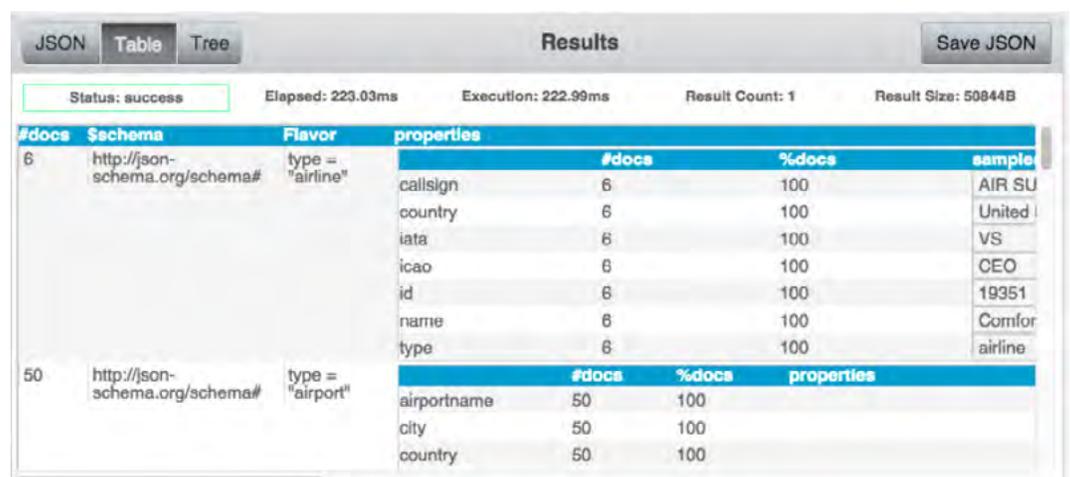
## Powerful Query Tools

### Query Workbench

There's nothing better than being free of a fixed schema, being able to modify the data model within your application as you see fit, when you see fit. However, that doesn't mean schemas are bad. While we don't want them to get in the way of development, we don't want them to be a mystery either.

There are times when we simply need to see the schema. We've all used Toad (or some other GUI) for connecting to a relational database and browsing its schema. After all, if you can't see it, you might as well be flying blind. There is no schema, right? Sort of.

Because JSON is self-describing, it's easy to infer a schema. All we have to do is sample the data, and that's exactly what we do. When you execute an INFER statement on a bucket within the Query Workbench, it will identify document types and elements, and display sample data for them. In addition, the bucket analyzer can be used to find out if buckets have full, limited, or no query support based on the indexes you've created. Mystery solved.

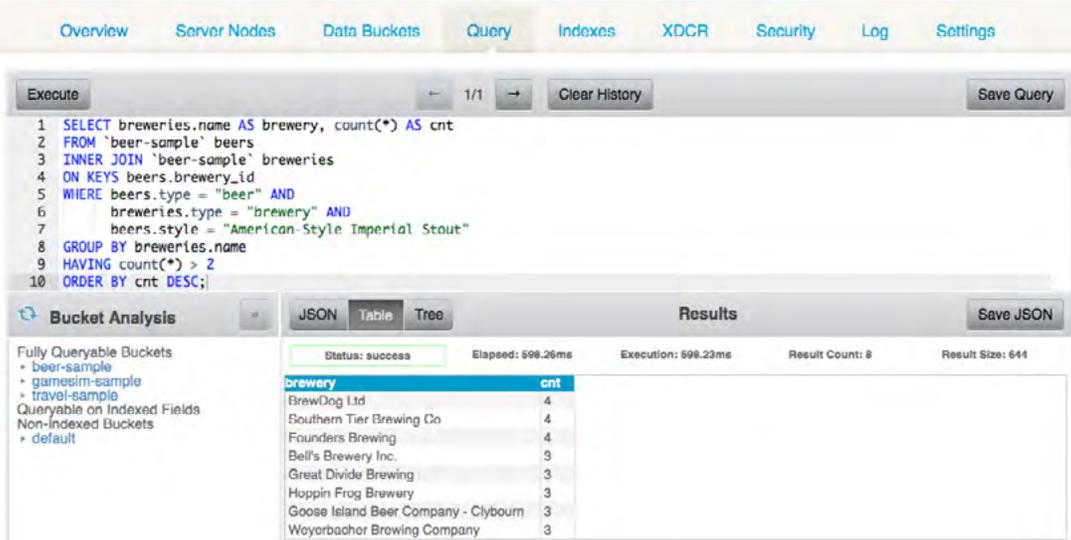


The screenshot shows the 'Results' window in Couchbase Query Workbench. It displays the results of an INFER query, showing two document types: 'airline' and 'airport'. Each type is represented by a table with columns for '#docs', '\$schema', 'Flavor', and 'properties'. The 'airline' type has 6 documents and lists properties like 'callsign', 'country', 'iata', 'icao', 'id', 'name', and 'type'. The 'airport' type has 50 documents and lists properties like 'airportname', 'city', and 'country'. A 'sample' column provides example values for each property.

#docs	\$schema	Flavor	properties																								
6	http://json-schema.org/schema#	type = "airline"	<table border="1"> <thead> <tr> <th>#docs</th> <th>%docs</th> <th>sample</th> </tr> </thead> <tbody> <tr> <td>6</td> <td>100</td> <td>AIR SU</td> </tr> <tr> <td>6</td> <td>100</td> <td>United</td> </tr> <tr> <td>6</td> <td>100</td> <td>VS</td> </tr> <tr> <td>6</td> <td>100</td> <td>CEO</td> </tr> <tr> <td>6</td> <td>100</td> <td>19351</td> </tr> <tr> <td>6</td> <td>100</td> <td>Comfor</td> </tr> <tr> <td>6</td> <td>100</td> <td>airline</td> </tr> </tbody> </table>	#docs	%docs	sample	6	100	AIR SU	6	100	United	6	100	VS	6	100	CEO	6	100	19351	6	100	Comfor	6	100	airline
#docs	%docs	sample																									
6	100	AIR SU																									
6	100	United																									
6	100	VS																									
6	100	CEO																									
6	100	19351																									
6	100	Comfor																									
6	100	airline																									
50	http://json-schema.org/schema#	type = "airport"	<table border="1"> <thead> <tr> <th>#docs</th> <th>%docs</th> <th>properties</th> </tr> </thead> <tbody> <tr> <td>50</td> <td>100</td> <td>airportname</td> </tr> <tr> <td>50</td> <td>100</td> <td>city</td> </tr> <tr> <td>50</td> <td>100</td> <td>country</td> </tr> </tbody> </table>	#docs	%docs	properties	50	100	airportname	50	100	city	50	100	country												
#docs	%docs	properties																									
50	100	airportname																									
50	100	city																									
50	100	country																									

We're not done yet. You've seen the schema, now it's time to write the queries. There's a time and place to code first, ask questions later. It's not when you're trying to write a query for the first time, or when you simply want to take a look the data. If you've used Eclipse or Visual Studio (or some other IDE) to create and run queries, you know that when it comes to NoSQL, you've been missing out. Not any more.

Within the Query Workbench, you can create, edit, save, and run queries, and it supports multi-line formatting, syntax highlighting, and autocomplete. You can even display query results as a document, table, or tree. It has everything you need to start writing queries. Life is about to get a lot easier.



The screenshot shows the Query Workbench interface with the following components:

- Navigation tabs: Overview, Server Nodes, Data Buckets, Query, Indexes, XDCR, Security, Log, Settings.
- Execute button and status: 1/1, Clear History, Save Query.
- SQL Query:

```
1 SELECT breweries.name AS brewery, count(*) AS cnt
2 FROM `beer-sample` beers
3 INNER JOIN `beer-sample` breweries
4 ON KEYS beers.brewery_id
5 WHERE beers.type = "beer" AND
6       breweries.type = "brewery" AND
7       beers.style = "American-Style Imperial Stout"
8 GROUP BY breweries.name
9 HAVING count(*) > 2
10 ORDER BY cnt DESC;
```
- Bucket Analysis sidebar showing a tree of buckets.
- Results section with a table view showing the following data:

brewery	cnt
BrewDog Ltd	4
Southern Tier Brewing Co	4
Founders Brewing	4
Bell's Brewery Inc.	3
Great Divide Brewing	3
Hoppin Frog Brewery	3
Goose Island Beer Company - Clybourn	3
Weyerbacher Browing Company	3

## Query Shell (CBQ)

Then again, maybe you prefer the command line. If you've taken sides in the editor war, Vi versus Emacs, the query shell (CBQ) is for you. It's not new, but we've made it more advanced.

With smart connection management, you can connect to the query service or to the cluster. Love your scripts? Well, it's fully scriptable now. You can create scripts to create indexes, insert data, execute queries, and more. Running Linux? Great, you can pipe commands to CBQ. Want to run commands saved in a file, or perhaps save the results to a file? Go for it.

You live and die by the command line. We get it.

## Query Monitoring

With great power, comes great responsibility, or, at the very least, the need to monitor queries and if necessary, terminate them. The last thing you need is a rogue query. That being said, there's more to it than monitoring the queries themselves.

We've introduced additional system catalogs (or keyspaces) for monitoring the query service too. You can monitor resource usage (e.g. memory/CPU) and performance (e.g. requests per second) as well as the use of prepared statements. With the active and completed request catalogs, you can identify long-running requests. For example, you can terminate a long-running query or identify the top N slowest queries.

You can monitor the query service and active/completed requests via REST or N1QL:

<b>QUERY SERVICE</b>	<b>REST</b>	GET <a href="http://localhost:8093/admin/vitals">http://localhost:8093/admin/vitals</a>
<b>ACTIVE REQUESTS</b>	<b>REST</b>	GET <a href="http://localhost:8093/admin/active_requests">http://localhost:8093/admin/active_requests</a>
	<b>N1QL</b>	SELECT * FROM system:active_requests;
<b>COMPLETED REQUESTS</b>	<b>REST</b>	GET <a href="http://localhost:8093/admin/completed_requests">http://localhost:8093/admin/completed_requests</a>
	<b>N1QL</b>	SELECT * FROM system:completed_requests;
<b>PREPARED STATEMENTS</b>	<b>REST</b>	GET <a href="http://localhost:8093/admin/prepareds">http://localhost:8093/admin/prepareds</a>
	<b>N1QL</b>	SELECT * FROM system:prepareds

## Faster Querying and Indexing

### Memory-optimized indexes

We like memory. It's fast. But what does that have to do with indexes? If an index can't fit in memory, we'll cache what we can. However, what if the index can fit in memory? Should we simply cache the whole thing? No, and here's why. If the index fits in memory, it doesn't have to be on disk, and if it doesn't have to be on disk, it doesn't have to use a disk-optimized data structure — the B-tree.

Wouldn't it better to stop maintaining the index on disk, and use a memory-optimized data structure? Everything would be faster. Index updates? Faster. Index scans? Faster. That's why we introduced a memory-optimized storage engine for the index service, and to make it 100% pure awesome, we used skip lists.

Now, I know what you're thinking. It's faster to maintain indexes in memory, but what if the index service is restarted? Nothing. You scheduled periodic snapshots to avoid rebuilding all of the indexes from scratch. Wait a minute. That only works if we write to disk. You're right. That's why we call them "memory-optimized" indexes, not "in-memory" indexes. That, and we're not caching blocks of file data in memory. We're using a memory-optimized data structure, taking snapshots, and writing them to disk.

We chose lock-free, in-memory skip lists because they're fast - we're talking  $O(\log n)$  - they can handle a lot of concurrent updates, and we wanted to remove disk IO from the critical path. With memory-optimized indexes, you can look forward to consistent, low-latency index operations — faster scans and faster updates for faster queries. It's that simple.

### Index joins

The only thing better than joins, is faster joins. Your basic left to right join? Not a problem. In the beer sample, included in Couchbase Server, beers (left side) reference breweries (right side). If you want to find all Imperial Stouts and the state they're brewed in, we'll find all Imperial Stouts (preferably with a secondary index on the "style" field), then we'll find their breweries (using the "brewery\_id" field), and finally, we'll perform the join — it's your standard lookup join, and it's fast.

But what happens when you want to find all beers brewed in California? We could find all beers, find their breweries, perform the join, and finally, scan the intermediate results, removing the ones where the brewery is not in California. Pretty inefficient. That's why we introduced index joins.

Now, about those beers brewed in California. How about we find all breweries in California (with a secondary index on the "state" field), find their beers (preferably with a secondary index on the "brewery\_id" field), and then perform the join? Yes, we know. It would be much faster. In fact, that's exactly how we do it now.

LOOKUP JOIN (LEFT TO RIGHT, LEFT SIDE FIRST)	INDEX JOIN (LEFT TO RIGHT, RIGHT SIDE FIRST)
<pre>SELECT * FROM `beer-sample` beer JOIN `beer-sample` brewery ON KEYS beer.brewery_id WHERE beer.type="beer" AND brewery.type="brewery" AND brewery.state="California"</pre>	<pre>SELECT * FROM `beer-sample` brewery JOIN `beer-sample` beer ON KEY beer.brewery_id FOR brewery WHERE beer.type="beer" AND brewery.type="brewery" AND brewery.state="California"</pre>

While lookup joins can use an index to filter the left-hand side first, before the join, index joins can use an index to filter right-hand side first, before the join. The join syntax has been extended so queries can define the right hand side first to perform an index join instead of a lookup join.

### Array Indexing

You have JSON. It has arrays. If only we could index them. After all, your queries would be faster and your indexes would be smaller. You win. You can now query documents with arrays by creating indexes on arrays of scalar values (string, number, etc) and objects. We'll toss in nested elements too, including arrays of arrays.

In the travel sample, included in Couchbase Server, airlines have routes, a route has a schedule, and a schedule has flights. Yes, the flight schedule is an array of flights. A flight is comprised of day, time, and flight number. If you create an index on the number of stops and the day of the week, you can create a query to find all nonstop flights on say, Sunday.

EXAMPLE ROUTE	INDEX	QUERY
<pre>{   "type": "route",   "airline": "AA",   "airlineid": "airline_24",   "sourceairport": "MCO",   "destinationairport": "SEA",   "stops": "0",   "equipment": "737",   "schedule": [     {"day": 1, "utc": "13:25:00", "flight": "AA788"},     {"day": 4, "utc": "13:25:00", "flight": "AA419"},     {"day": 5, "utc": "13:25:00", "flight": "AA519"}   ] }</pre>	<pre>CREATE INDEX flight_day_stops ON `travel-sample` (stops, DISTINCT ARRAY flight.day FOR flight IN schedule) WHERE type = "route"</pre>	<pre>SELECT * FROM `travel-sample` WHERE type = "route" AND stops = "0" AND ANY flight IN schedule SATISFIES flight.day = "1"</pre>

### Read Your Own Writes (RYOW)

First things first, we're talking about query consistency. CRUD operations (e.g. read/write) have always been — and shall always be — strongly consistent. However, when it comes to queries, you've had two options: eventually consistent and strongly consistent. By default, secondary indexes are updated asynchronously to maintain low-latency writes. As a result, queries are eventually consistent, but they're really fast. Now, you can specify strong consistency if you like, but your queries will be slower. After all, the indexes have to be updated first.

You shouldn't have to choose between consistency and performance. We want you to have your cake and eat it too. What if you could get most of the performance and most of the consistency rather than one or the other? Well, that's what you get with read-your-own-writes consistency. When specified, indexes will be updated to be as current as the client's last write. Your queries will be faster. Your results will be more consistent. It's the best of both worlds.

## Circular writes

If your indexes can't fit in memory, we'll continue to use ForestDB — our next generation storage engine. However, even in cases where the indexes have to be maintained on disk, we wanted to improve query performance.

With circular writes, we not only improve query performance, we reduce disk IO and the size of index files. It's pretty simple. When writing changes to disk, we check to see if there are any stale blocks available — existing blocks that contain stale data or are otherwise unused. If there are, we'll write to them. If not, we'll create new blocks and append them at the end of the file. Also with circular writes, index files grow slower and that means we don't have to perform compaction as often.

We left no rock unturned because when it comes to performance and efficiency, every little bit counts.

## Fine-grained data access

### Sub-document API

We know, we know. Partial updates has been a missing feature. Why the wait? Like everything else, we wanted to do it right. We wanted to come up with an elegant solution, and that's exactly what the sub-document API is. However, it's not limited to partial updates. With the sub-document API, you can read and write specific fields rather than reading and writing whole documents. Awesome. The last thing you want to do is read and write a whole document just so you can update a single field.

It's not only more elegant, it's much faster. Especially if you have large documents and/or arrays. There's a lot less network overhead — no need to transmit whole documents over the wire. It will make your life a lot easier, too. What if you need to update a document but you don't have it? No worries. You don't need it. What if you need multiple clients to update different fields at the same time? No problem. Go for it. You no longer need to use compare-and-swap (CAS).

Let's take a look at the Java API for reading and writing specific fields.

### READ OPERATIONS

<b>[Field] Exists</b>	<code>bucket.lookupIn(id).exists(path).execute();</code>
<b>[Field] Get</b>	<code>bucket.lookupIn(id).get(path).execute();</code>

### WRITE OPERATIONS

<b>[Field] Insert</b>	<code>bucket.mutateIn(id).insert(path, value, createParents).execute();</code>
<b>[Field] Remove</b>	<code>bucket.mutateIn(id).remove(path).execute();</code>
<b>[Field] Replace</b>	<code>bucket.mutateIn(id).replace(path, value).execute();</code>
<b>[Field] Insert or replace</b>	<code>bucket.mutateIn(id).upsert(path, value, createParents).execute();</code>
<b>[Array] Add a unique element</b>	<code>bucket.mutateIn(id).arrayAddUnique(path, value, createParents).execute();</code>
<b>[Array] Add an element at the back</b>	<code>bucket.mutateIn(id).arrayAppend(path, value, createParents).execute();</code>
<b>[Array] Add elements at the back</b>	<code>bucket.mutateIn(id).arrayAppendAll(path, values, createParents).execute();</code>
<b>[Array] Add elements at the back</b>	<code>bucket.mutateIn(id).arrayAppendAll(path, values).execute();</code>
<b>[Array] Add element at a position</b>	<code>bucket.mutateIn(id).arrayInsert(path, value).execute();</code>
<b>[Array] Add elements at a position</b>	<code>bucket.mutateIn(id).arrayInsertAll(path, values).execute();</code>
<b>[Array] Add elements at a position</b>	<code>bucket.mutateIn(id).arrayInsertAll(path, values).execute();</code>
<b>[Array] Add an element at the front</b>	<code>bucket.mutateIn(id).arrayPrepend(path, value, createParents).execute();</code>
<b>[Array] Add elements at the front</b>	<code>bucket.mutateIn(id).arrayPrependAll(path, values, createParents).execute();</code>
<b>[Array] Add elements at the front</b>	<code>bucket.mutateIn(id).arrayPrependAll(path, values).execute();</code>
<b>[Counter] Increment or decrement</b>	<code>bucket.mutateIn(id).counter(path, delta, createParents).execute();</code>

Wait a minute. Is that a builder? Why, yes it is. You can perform multiple read or write operations at the same time:

```
bucket.lookupIn("user::shane")
  .exists("twitter")
  .get("job.company")
  .execute();

{
  "firstName": "Shane",
  "lastName": "Johnson",
  "twitter": "shane_dev",
  "skills": [
    "Java", "Distributed Systems"],
  "job": {
    "company": "Couchbase"
    "role": "Product Marketing"}
  "friends": 1
}

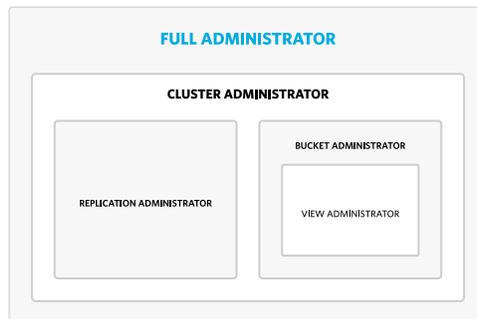
bucket.mutateIn("user::shane")
  .insert("middleInitial", "K", true)
  .arrayAddUnique("skills", "Marketing", true)
  .replace("job.role", "Chef")
  .counter("friends", 1, true)
  .execute();
```

## Advanced security

### Role-based Access Control for Administrators

Not everyone needs full administrative access. The fewer, the better. Ideally, you should be able to have different administrators perform different functions, and with little overlap. The goal being to separate and restrict administrative access as much as possible. That’s why we’ve created six out-of-the-box administrative roles: full, cluster, bucket, view, replication, and read-only.

The administrative roles are hierarchical with nested roles being more restricted.



### X.509 Certificates

We’ve long supported encrypted communication, but we’re taking it further with support for x.509 certificates. It not only lets you use your own certificate authority (CA), internal or third-party, it simplifies certificate management and rotation. To top it off, we support both single-tier and n-tier CA hierarchies — allowing you to use separate root, intermediate, and issuing certificate authorities.

## Improved disaster recovery

### Enterprise Backup and Restore

It’s one thing to backup a few gigabytes of data, it’s another to backup terabytes of data. If you recognize terms like Recovery Time Objective (RTO) and Recovery Point Objective (RPO), keep reading — we’ve created a separate backup/restore tool just for you. It leverages Database Change Protocol (DCP) streams and multiple threads to backup and restore data faster, much faster.

What if the backup fails? Not a problem. It is capable of resuming from where it left off. What if the data is being rebalanced? We’ve got you covered. It recognizes when data is moving, and will follow it. What if one or more nodes fail? We considered that, too. It will log the failed nodes and continue to backup data on the running nodes. When the failed nodes are restored, the backup can be resumed and it will backup the data on the previously failed nodes.

Now about those backups. The first time around, a full backup will be taken. However, subsequent backups will be incremental — much faster. All backups will be stored in a Backup Repository, and

all repositories will be stored in a Backup Archive. This isn't your basic backup/restore tool. Within a backup repository, the data is stored in a separate database (ForestDB) while scripts (e.g. create index), configuration, and metadata are stored as JSON files.

## Extended platform support

### Docker and Red Hat OpenShift

Yes, the time has come. We've supported containers in development and test environments for a while. For the past couple of years, more and more of our customers have been moving to containers, and they've been requesting production support. The wait is over. After rigorous testing, including regression, functional, system, volume, upgrade, and of course, performance, we've certified Docker and Red Hat OpenShift for production.

### Debian 8

By the way, we support Debian 8 now.

## Developer preview features

### Integrated full-text Search

We know you've been waiting for it, and here it is. We've implemented full-text search on top of the popular open source project, Bleve. Written in Go, Bleve supports text analysis, faceting, scoring, boosting, and highlighting with term, fuzzy, phrase, match / match phrase, prefix, and regex / wildcard queries. Why stop there? How about compound queries with conjunction, disjunction, and boolean combinations? Yes, you can do that, too. And we tossed in range queries, too.

To top it off, full-text search indexes are partitioned and replicated. That's right. You get highly scalable, highly available text indexes. With the addition of full-text search, you now have complete access to your data — read/write, query (SQL), geospatial, views (MapReduce), and full-text search.

That being said, we're not quite done. So, we're introducing full-text search as a developer preview feature. We still have a little bit of work left to do, but we wanted you to be able to try it out. Let us know how it goes!

## About Couchbase

Couchbase delivers the database for the Digital Economy. Developers around the world choose Couchbase for its advantages in data model flexibility, elastic scalability, performance, and 24x365 availability to build enterprise web, mobile, and IoT applications. The Couchbase platform includes Couchbase, Couchbase Lite - the first mobile NoSQL database, and Couchbase Sync Gateway. Couchbase is designed for global deployments, with configurable cross data center replication to increase data locality and availability. All Couchbase products are open source projects. Couchbase customers include industry leaders like AOL, AT&T, Cisco, Comcast, Concur, Disney, DIXONS, eBay, General Electric, Marriott, Nordstrom, Neiman Marcus, PayPal, Ryanair, Rakuten / Viber, Tesco, Verizon, Wells Fargo, as well as hundreds of other household names. Couchbase investors include Accel Partners, Adams Street Partners, Ignition Partners, Mayfield Fund, North Bridge Venture Partners, Sorenson Capital and WestSummit Capital.



2440 West El Camino Real | Ste 600  
Mountain View, California 94040

1-650-417-7500

[www.couchbase.com](http://www.couchbase.com)