Apache Cassandra

A NoSQL Database

What is Cassandra?

- The Apache Cassandra is a NoSQL database which provides scalability and high availability without compromising performance.
- Linear scalability and proven fault-tolerance on commodity hardware make it the perfect platform for mission-critical data.
- Cassandra's support for replicating across multiple data centers is best-in-class.

A Brief History

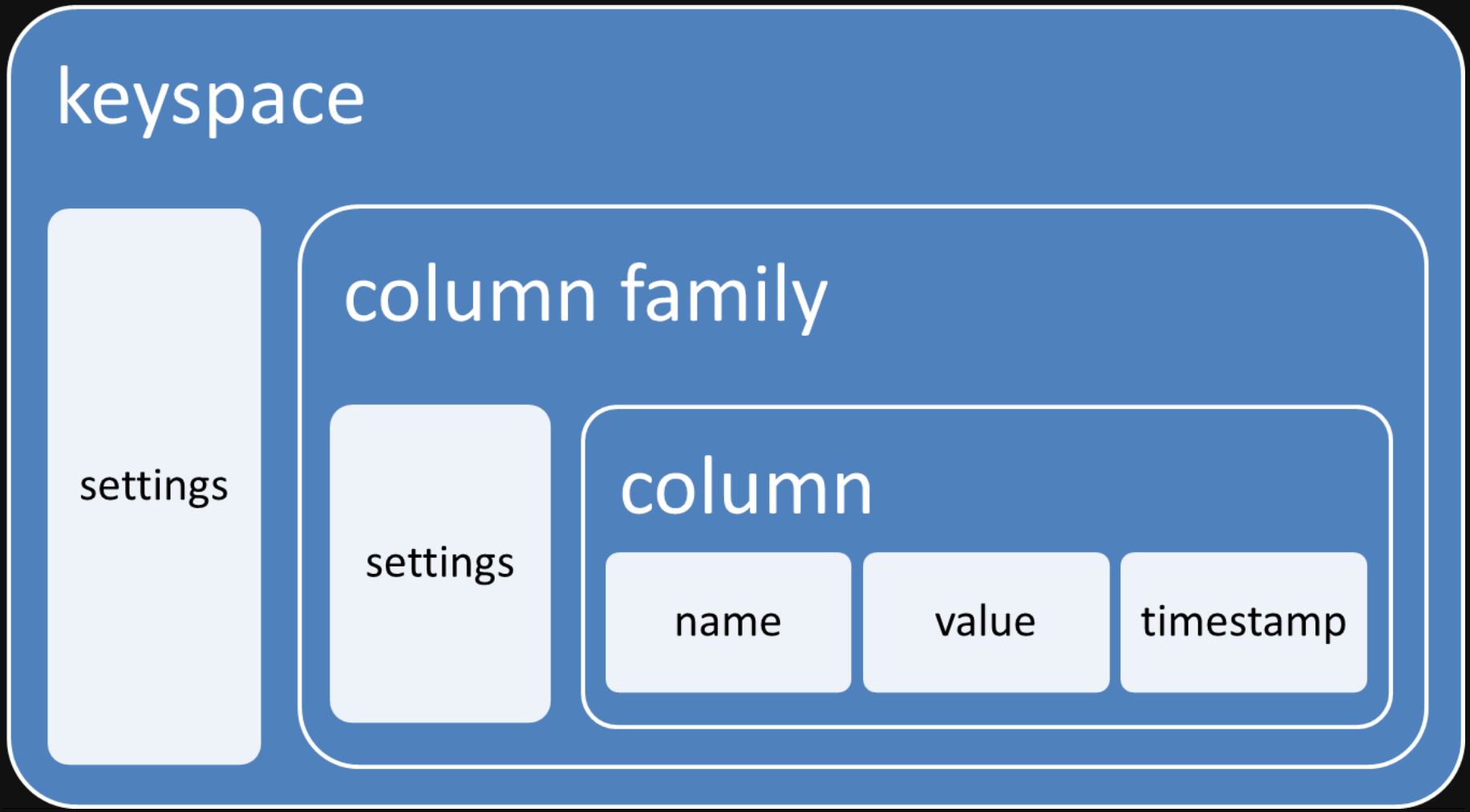
- Initially developed at Facebook for inbox search.
- Open sourced by Facebook in 2008.
- Accepted into Apache Incubator in 2009.
- Apache top-level project since February 2010.
- Now maintained by the Apache Software Foundation.

Features

- **Fault Tolerant**
  - Data is automatically replicated to multiple nodes for fault-tolerance.
  - Failed nodes can be replaced with no downtime.
- **Performant**
  - Cassandra consistently outperforms popular NoSQL alternatives in benchmarks.
- **Decentralized**
  - There are no single points of failure. There are no network bottlenecks. Every node in the cluster is identical.

- **Scalable**
  - Some of the largest production deployments include:
  - Apple's, with over 75,000 nodes storing over 10 PB of data,
  - Netflix (2,500 nodes, 420 TB, over 1 trillion requests per day),
  - eBay (over 100 nodes, 250 TB).
- **Durable**
  - Cassandra is suitable for applications that can't afford to lose data, even when an entire data center goes down.

Data Model


keyspace
settings
column family
settings
column
name
value
timestamp

Data Model

- Keyspace
  - A keyspace defines a number of options that applies to all the tables it contains.
  - Most prominent is the replication strategy.
- Table
  - A multi-dimensional map indexed by key which stores the rows.
- Column
  - Basic data structure that stores the data elements.
  - Each column contains name, value and timestamp.

Cassandra Query Language (CQL)

- Offers a syntax similar to SQL.
- Offers a data model similar to SQL so that data is stored in rows of columns.
- Provides:
  - Data Definition Statements
  - Data Manipulation Statements
- Using the command **`cqlsh`**.

Type	Description
blob	Arbitrary Bytes
boolean	*True* or *False* values
date	Date (without time value)
timestamp	Date (with time value)
double	64-bit Floating Point
float	32-bit Floating Point
int	32-bit Signed Integer
text	UTF8 Encoded String
varchar	UTF8 Encoded String

Type	Description
map<T, T>	A sorted set of key-value pairs
set<T>	A sorted collection of unique values
list<T>	A collection of non-unique values
tuple<T, T>	A pair of values (a, b)

Creating a Keyspace

```
CREATE KEYSPACE Emp
   WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 1};
```

- Creates a keyspace named **Emp**
- replication_factor determines number of copies.

Switch to created keyspace using:

```
USE Emp;
```

Creating a Table

```
CREATE TABLE [table name] (
      [column name] [column type] [PRIMARY KEY]
      ) WITH [table options];
```

Example

```
CREATE TABLE Employee (
      id int PRIMARY KEY,
      name text,
      city text,
      salary int);
```

Insert Statement

```
INSERT INTO [table name] ([column names])
                VALUES ([values]);
```

Example

```
INSERT INTO Employee (id, name, city, salary)
            VALUES (1, 'Name 1', 'Pune', 25000);
```

- If the row with given primary key does not exist, it will be created.
- Otherwise, the existing row will be updated.

Select Statement

```
SELECT [column names] FROM [table]
                WHERE [where clause]
                ORDER BY [column name [ASC | DESC]]
                LIMIT [limit];
```

Example

```
SELECT id, name, city, salary FROM Employee;
```

Update Statement

```
UPDATE [table name]
        SET [column name] = [value]
        WHERE [where clause];
```

Example

```
UPDATE Employee SET salary = 50000 WHERE id = 4;
```

Delete Statement

```
DELETE FROM [table name] WHERE [where clause];
```

Example

```
DELETE FROM Employee WHERE id = 1;
```

- If column name is specified after DELETE keyword, only that column is removed.
- Otherwise the entire row is removed.

Aggregate Functions

- Aggregate functions work on a set of rows to deliver an aggregated result.
- Following built in aggregation functions are available:
  - **`count()`**: Count rows returned by the query.
  - **`max()` and `min()`**: Compute maximum and minimum value returned by a query.
  - **`sum()`**: Sum up all values returned by a query.
  - **`avg()`**: Average of all values returned by a query.

Aggregation Examples

Count:

```
SELECT count(*) FROM Employee;
```

Min and Max:

```
SELECT max(salary) FROM Employee;
SELECT min(salary) FROM Employee;
```

Sum:

```
SELECT sum(salary) FROM Employee WHERE city = 'Pune';
```

Avg:

```
SELECT avg(salary) FROM Employee WHERE city = 'London';
```

Indexing

- Indexed using primary key by default.
- Secondary indexes can be created as required with the syntax:

```
CREATE INDEX [index name] on [table name] ([column]);
```

```
CREATE INDEX nameIndex on Employee (name);
```

- Indexes can be dropped by using:

```
DROP INDEX [index name];
```

```
DROP INDEX nameIndex;
```