Oracle NoSQL Database

Compared to Cassandra and HBase

Overview

- Oracle NoSQL Database server is licensed under AGPL and the client is licensed under Apache 2.0. Licensing the client under 2.0 allows other open sources products to freely integrate and distributed an Oracle NoSQL driver with their product.
- Cassandra and Hbase– both client and server are Apache 2.0 licensed.
- Oracle NoSQL Database is a scalable key-value store for data represented in a typed Table format, JSON format and with an application extension in Graph format. Oracle NoSQL Database most closely resembles competitive solutions such as Cassandra, Hbase, DynamoDB, Riak.
- Oracle NoSQL Database is designed to provide extreme scale OLTP type storage and retrieval for hierarchical data structures – logically and physically co-located hierarchical relations, but no system wide JOIN relations. Data storage provides flexible durability on a per operation basis, from cache based eventual consistency to proper ACID transactions. Data retrieval is by key and/or secondary index over values. Query is capable of range based predicates and system wide ordered results and operations are done in parallel while providing flexible data consistency guarantees.

  Oracle NoSQL Database uses a distributed architecture which scales data (and processing) out on commodity servers using a hashing algorithm and intelligent client drivers. Oracle NoSQL Database uses a PAXOS leader as the coordinator for data replication and transactions within a range of values based on the hashing algorithm and the current size of the cluster. This is similar to when using a new client driver from Cassandra, called a “Token Aware” driver which intends to improve the performance of Cassandra by directing client requests to a process in a data range that is known to have the data. This intelligent routing is an inherent part of the Oracle NoSQL Database design and provides for automated topology management and out of the box load balancing of requests.

  Cassandra uses consistent hashing over a peer-to-peer architecture where every node in the system can handle any read-write request, so arbitrary nodes become coordinators of requests when they do not actually hold the data involved in the request operation. That means both an extra network hop (minimum) for each call and it means the failure of a single node can have system wide performance impacts as other arbitrary nodes change their behavior in response to

the failed node. Using a PAXOS leader based high availability and durability design, Oracle NoSQL Database is not vulnerable to the system wide cascading failures found in peer-to-peer systems like Cassandra.

HBase uses a master-slave distribution architecture which hashes data values into master processes known as “Region Servers” each of which are responsible for some set of ranges of the key space. Those region servers write the data (thru several layers of indirection) to HDFS (hadoop distributed file system) which handles data recoverability thru file system replication. The Region servers also make the data available to 1 other process which can serve read requests. This is distinctly different from Oracle NoSQL Database which allows the end user to determine how many read replicas are needed to meet their concurrency requirements and uses memory/process based replication for high availability rather than purely disk based measures.

Similar to Cassandra and Hbase, Oracle NoSQL Database provides a Table based meta model capable of parent child relationships to deal with hierarchical data. This provides a person familiar with relational databases a data modeling paradigm in which they can apply many of their existing skills.

Hbase does not support Data Center deployments and failover. Both Oracle NoSQL and Cassandra provide for automated failover in Data Center deployments using a “one cluster” distribution model. Cassandra implements this using what is called a “snitch” that lays out the location of the copies distributed across all the data centers. Oracle NoSQL implements this using a smart topology driver that works with PAXOS groups the server deploys in a highly available manner across the data centers. A data center (zone) can have independent specifications for the number of data copies it will hold and on failure of any given data center, the remaining data centers will automatically take over for read-write operations. In Oracle NoSQL, low latency of read requests in a data center deployment are facilitated by latency aware client drivers, ensuring that requests are always sent to the fastest responding node with the shortest work queue. In both systems, low latency write requests can be enabled by reducing levels of durability guarantee.

Comparison

The table below gives a high level comparison of Oracle NoSQL Database and Cassandra features/capabilities. Low level details are found in links to Oracle and Cassandra online documentation.

Point	HBase	Cassandra	ONDB
Foundations	HBase is based on BigTable (Google)	Cassandra is based on DynamoDB (Amazon). Initially developed at Facebook by former Amazon engineers. This is one reason why Cassandra supports multi data center.	ONDB is based Oracle Berkeley DB Java Edition a mature log-structured, high performance, transactional database.
Infrastructure	HBase uses the Hadoop Infrastructure (Zookeeper, NameNode, HDFS). Organizations that will deploy Hadoop anyway may be comfortable with leveraging Hadoop knowledge by using HBase	Cassandra started and evolved separate from Hadoop and its infrastructure and Operational knowledge requirements are different than Hadoop. However, for analytics, many Cassandra deployments use Cassandra + Storm (which uses Zookeeper), and/or Cassandra + Hadoop.	ONDB has simple infrastructure requirements and does not use Zookeeper. Hadoop based analytics are supported via a ONDB/Hadoop connector.
Infrastructure Simplicity and SPOF	The HBase-Hadoop Infrastructure has several "moving parts" consisting of Zookeeper, Name Node, Hbase Master, and Data Nodes, Zookeeper is	Cassandra uses a a single Node-type. All nodes are equal and perform all functions. Any Node can act as a coordinator, ensuring no SPOF. Adding Storm or Hadoop, of course, adds complexity to	ONDB uses a single node type to store data and satisfy read requests. Any node can accept a request and forward it if necessary. There is no SPOF. In addition, there is a simple watchdog process (the Storage Node Agent or SNA for short) on each machine to ensure high availability and automatically restart any data storage node in case of process level failures.

	clustered and naturally fault tolerant. Name Node needs to be clustered to be fault tolerant.	the infrastructure.	The SNA also helps with administration of the store.
Read Intensive Use Cases	HBase is optimized for reads, supported by single-write master, and resulting strict consistency model, as well as use of Ordered Partitioning which supports row-scans. HBase is well suited for doing Range based scans.	Cassandra has excellent single-row read performance as long as eventual consistency semantics are sufficient for the use-case. Cassandra quorum reads, which are required for strict consistency will naturally be slower than Hbase reads. Cassandra does not support Range based row-scans which may be limiting in certain use-cases. Cassandra is well suited for supporting single-row queries, or selecting multiple rows based on a Column-Value index.	ONDB provides: 1) Strict consistency reads at the master 2) eventual consistency reads, with optional time constraints on the recency of data and 3) application level *Read your writes* consistency. All reads contact just a single storage node making read operations very efficient. ONDB also supports range based scans.
Multi-Data Center Support and Disaster Recovery	HBase provides for asynchronous replication of an HBase Cluster across a WAN. HBase clusters cannot be set up to achieve zero RPO, but in steady-state HBase should be roughly failover-equivalent to any other DBMS that relies on asynchronous	Cassandra Random Partitioning provides for row-replication of a single row across a WAN, either asynchronous (write.ONE, write.LOCAL_QUORUM), or synchronous (write.QUORUM, write.ALL). Cassandra clusters can therefore be set up to achieve zero	Release 3.0 provides for asynchronous cascaded replication across data centers.

	replication over a WAN. Fall-back processes and procedures (e.g. after failover) are TBD.	RPO, but each write will require at least one wan-ACK back to the coordinator to achieve this capability.	
Write.ONE Durability	Writes are replicated in a pipeline fashion: the first-data-node for the region persists the write, and then sends the write to the next Natural Endpoint, and so-on in a pipeline fashion. HBase's commit log "acks" a write only after *all* of the nodes in the pipeline have written the data to their OS buffers. The first Region Server in the pipeline must also have persisted the write to its WAL.	Cassandra's coordinators will send parallel write-requests to all Natural Endpoints, The coordinator will "ack" the write after exactly one Natural Endpoint has "acked" the write, which means that node has also persisted the write to its WAL. The writes may or may not have committed to any other Natural Endpoint.	ONDB considers a request with ReplicaAckPolicy.NONE (the ONDB equivalent of Write.ONE) as having completed after the change has been written to the master's log buffer; the change is propagated to the other members of the replication group, via an efficient asynchronous stream-based protcol.
Ordered Partitioning	HBase only supports Ordered Partitoning. This means that Rows for a CF are stored in RowKey order in HFiles, where each Hfile contains a "block" or "shard" of all the rows in a CF. HFiles are distributed across all data-nodes in the Cluster	Cassandra officially supports Ordered Partitioning, but no production user of Cassandra uses Ordered Partitioning due to the "hot spots" it creates and the operational difficulties such hot-spots cause. Random Partitioning is the only recommended Cassandra partitioning scheme, and rows are	ONDB only supports random partitioning. Prevailing experience indicates that other forms of partioning are really hard to administer in practice.

		distributed across all nodes in the cluster.	
RowKey Range Scans	Because of ordered partitioning, HBase queries can be formulated with partial start and end row-keys, and can locate rows inclusive-of, or exclusive of these partial-rowkeys. The start and end row-keys in a range-scan need not even exist in Hbase.	Because of random partitioning, partial rowkeys cannot be used with Cassandra. RowKeys must be known exactly. Counting rows in a CF is complicated. It is highly recommended that for these types of use-cases, data should be stored in columns in Cassandra, not in rows.	ONDB range requests can be defined with partial start and end row-keys. The start and end row-keys in a range-scan need not exist in the store.
Linear Scalability for large tables and range scans	Due to Ordered Partitioning, HBase will easily scale horizontally while still supporting rowkey range scans.	If data is stored in columns in Cassandra to support range scans, the practical limitation of a row size in Cassandra is 10's of Megabytes. Rows larger than that causes problems with compaction overhead and time.	There are no limits on range scans across major or minor keys. Range scans across major keys require access to each shard in the store. Release 3 will support major key and index range scans that are parallelized across all the nodes in the store. Minor key scans are serviced by the single shard that contains the data associated with the minor key range.
Atomic Compare and Set	HBase supports Atomic Compare and Set. HBase supports supports transaction within a Row.	Cassandra does not support Atomic Compare and Set. Counters require dedicated counter column-families which because of eventual-consistency requires that all replicas in all natural end-points be read and updated with ACK. However, hinted-handoff mechanisms can make even these	ONDB supports atomic compare and set, making it simple to implement counters. ONDB also supports atomic modification of multiple minor key/value pairs under the same major key.

		built-in counters suspect for accuracy. FIFO queues are difficult (if not impossible) to implement with Cassandra.	
Read Load Balancing - single Row	Hbase does not support Read Load Balancing against a single row. A single row is served by exactly one region server at a time. Other replicas are used ony in case of a node failure. Scalability is primarily supported by Partitioning which statistically distributes reads of different rows across multiple data nodes.	Cassandra will support Read Load Balancing against a single row. However, this is primarily supported by Read.ONE, and eventual consistency must be taken into consideration. Scalability is primarily supported by Partitioning which distributes reads of different rows across multiple data nodes.	ONDB supports read load balancing. Only absolute consistency reads need to be directed to the master, eventual consistency reads may be served by any replica that can satisfy the read consistency requirements of the request.
Bloom Filters	Bloom Filters can be used in HBase as another form of Indexing. They work on the basis of RowKey or RowKey+Column Name to reduce the number of data-blocks that HBase has to read to satisfy a query. (Bloom Filters may exhibit false-positives (reading too much data), but never false negatives (reading	Cassandra uses bloom filters for key lookup.	Bloom filters are used to minimize reads to SST files that do not contain a requested key, in LSM-tree based storage underlying HBase and Cassandra. There is no need to create and maintain Bloom filters in the log-structured storage architecture used by ONDB.

	not enough data).		
Triggers	Triggers are supported by the CoProcessor capability in HBase. They allow HBase to observe the get/put/delete events on a table (CF), and then execute the trigger-logic. Triggers are coded as java classes.	Cassandra does not support co-processor-like functionality (as far as we know)	ONDB does not support triggers.
Secondary Indexes	Hbase does not natively support secondary indexes, but one use-case of Triggers is that a trigger on a "put" can automatically keep a secondary index up-to-date, and therefore not put the burden on the application (client).	Cassandra supports secondary indexes on column families where the column name is known. (Not on dynamic columns). Cassandra indexes are not stored locally and as a consequence are not well suited to low cardinality result sets on large numbers of records. Due to lack of transactions, indexes are updated independently of data and therefore become inconsistent over time and presumably need to be rebuilt.	Release 3.0 supports secondary indexes. The indexes are locally stored on the shard that contains the data being indexed. Indexes are maintained transactionally with data insert and update, so there is no divergence of the index. ONDB supports low cardinality result sets on extremely large numbers of records.
Simple Aggregation	Hbase CoProcessors support out-of-the-box simple aggregations in HBase. SUM, MIN, MAX, AVG, STD. Other aggregations can	Aggregations in Cassandra are not supported by the Cassandra nodes - client must provide aggregations. When the aggregation requirement spans multiple rows,	Aggregation is not supported by ONDB.

	be built by defining java-classes to perform the aggregation	Random Partitioning makes aggregations very difficult for the client. Recommendation is to use Storm or Hadoop for aggregations.	
HIVE Integration	HIVE can access HBase tables directly (uses de-serialization under the hood that is aware of the HBase file format).	Cassandra supports Hive query	ONDB supports Hive query.
PIG Integration	PIG has native support for writing into/reading from HBase.	Cassandra 0.7.4+	No PIG integration currently
CAP Theorem Focus	Consistency, Availability	Availability, Partition-Tolerance	Consistency, Availability, Limited Partition-Tolerance if there is a simple majority of nodes on one side of a partition (https://sleepycat.oracle.com/trac/wiki/JEKV/CAP has a detailed discussion) .
Consistency	Strong	Eventual (Strong is Optional)	Offers different read consistency models: 1) strict consistency reads at the master 2) eventual consistency reads, with optional time constraints on the recency of data and 3) Read your writes consistency.
Single Write Master	Yes	No (R+W+1 to get Strong Consistency)	Yes
Optimized For	Reads	Writes	Both reads and writes. Log-structured storage permits append-only writes, with each change being written once to disk. Reads can be serviced at any replica based upon the read consistency requirements associated with the request. Reads can be satisfied at a single node, by a single request to disk. There are no bloom filters to maintain and no risk of false positives causing

			multiple disk reads.
Main Data Structure	CF, RowKey, Name Value Pair Set	CF, RowKey, Name Value Pair Set	Major key, or minor key with its associated value.
Dynamic Columns	Yes	Yes	Provides equivalent functionality. Multiple minor keys can be dynamically associated with a major key.
Column Names as Data	Yes	Yes	Provides equivalent functionality via minor keys, which can be treated as data.
Static Columns	No	Yes	[R3.0 will support static columns]
RowKey Slices	Yes	No	No
Static Column Value Indexes	No	Yes	Yes
Sorted Column Names	Yes	Yes	Yes
Cell Versioning Support	Yes	No	No
Bloom Filters	Yes	Yes(only on Key)	Not necessary for ONDB
CoProcessors	Yes	No	No
Triggers	Yes(Part of Coprocessor)	No	No
Push Down Predicates	Yes(Part of Coprocessor)	No	No
Atomic Compare and Set	Yes	No	Yes
Explicit Row Locks	Yes	No	No
Row Key Caching	Yes	Yes	Yes
Partitioning Strategy	Ordered Partitioning	Random Partitioning recommended	Random partitioning

Rebalancing	Automatic	Not Needed with Random Partitioning	Not Needed with Random Partitioning
Availability	N-Replicas across Nodes	N-Replicas across Nodes	N-Replicas across Nodes
Data Node Failure	Graceful Degredation	Graceful Degredation	Graceful Degradation, as described in the availability section.
Data Node Failure - Replication	N-Replicas Preserved	(N-1) Replicas Preserved + Hinted Handoff	(N-1) Replicas Preserved.
Data Node Restoration	Same as Node Addition	Requires Node Repair Admin-action	Node catches up automatically by replaying changes from a member of the replication group.
Data Node Addition	Rebalancing Automatic	Rebalancing Requires Token-Assignment Adjustment	New nodes are added through the Admin service, which automatically redistributes data across the new nodes.
Data Node Management	Simple (Roll In, Role Out)	Human Admin Action Required	Human Admin action required.
Cluster Admin Nodes	Zookeeper, NameNode, HMaster	All Nodes are Equal	ONDB has a highly available Admin service, for administrative actions, eg. adding new nodes, replacing failed nodes, software updates, etc. but is not required for steady state operation of the service. There is a light weight SNA process(described earlier) on each machine to ensure high availability and restart any data storage node in case of failure.
SPOF	Now, all the Admin Nodes are Fault Tolerant	All Nodes are Equal	There is no SPOF, as described in the availability section.
Write.ANY	No, but Replicas are Node Agnostic	Yes (Writes Never Fail if this option is used)	No
Write.ONE	Standard, HA, Strong Consistency	Yes (often used), HA, Weak Consistency	Yes. Requires that the Master be reachable.
Write.QUORUM	No (not required)	Yes (often used with Read.QUORUM for Strong Consistency	Yes. This is the default.
Write.ALL	Yes (performance penalty)	Yes (performance penalty, not HA)	Yes (performance penalty, not HA)

Asynchronous WAN Replication	Yes, but it needs testing on corner cases.	Yes (Replica's can span data centers)	Asynchronous replication is routine in ONDB. Nodes local to the master will typically keep up, and nodes seperated by high latency WANs will have the changes replayed asynchronously via an efficient stream based protocol.
Synchronous WAN Replication	No	Yes with Write.QUORUM or Write.EACH-QUORUM	Yes, for requests that require acknowledgements (ReplicaAckPolicy.SIMPLE_MAJORITY or ReplicaAckPolicy.ALL). The acknowledging nodes will be synchronized with the master.
Compression Support	Yes	Yes	No