In-Memory Databases and Apache Ignite

Joan Tiffany To Ong Lopez 000457269 jtonglopez@gmail.com
Sergio José Ruiz Sainz 000458874 sergiers@opendeusto.es

18 December 2017
Table of Contents

1 Introduction ........................................................................................................................................... 5

2 Apache Ignite ......................................................................................................................................... 5
   2.1 Clustering ........................................................................................................................................ 5
   2.2 Durable Memory and Persistence .................................................................................................. 6
   2.3 Data Grid ......................................................................................................................................... 7
   2.4 Distributed SQL ............................................................................................................................ 10
   2.5 Compute Grid features .................................................................................................................. 10
   2.6 Other interesting features ............................................................................................................. 10

3 Business domain .................................................................................................................................. 11

4 Database schema and data setup ........................................................................................................ 11

5 Apache Ignite Walkthrough ................................................................................................................ 13
   5.1 Environment Setup ...................................................................................................................... 13
   5.2 Cluster/Node startup ..................................................................................................................... 13
   5.3 Domain Model generation ............................................................................................................ 15
   5.4 Loading caches ............................................................................................................................. 23
   5.5 Computing cache data .................................................................................................................. 24

6 Queries .................................................................................................................................................. 25
   6.1 Query 1: Multiple operations in a single procedure (generate invoices) ...................................... 25
   6.2 Query 2: Aggregate by sum according to a date range (compute revenue) ............................... 29
   6.3 Query 3: Update all rows of a single column (apply discount) .................................................... 30

7 Benchmark results ................................................................................................................................ 30
   7.1 Query 1 result ............................................................................................................................... 30
   7.2 Query 2 result ............................................................................................................................... 32
   7.3 Query 3 result ............................................................................................................................... 33
   7.4 Additional: Ignite cache loading .................................................................................................. 33

8 Development experience ..................................................................................................................... 34
   8.1 Open source vs closed source ...................................................................................................... 34
   8.2 Developer community ................................................................................................................... 35
   8.3 Developer support ....................................................................................................................... 36
   8.4 Documentation ............................................................................................................................. 37
   8.5 Tooling........................................................................................................................................... 37

9 Apache Ignite vs. other in-memory distributed databases ................................................................. 39
   9.1 Feature comparisons .................................................................................................................... 39
9.2 Performance comparisons ........................................................................................................... 39

10 Conclusion ................................................................................................................................. 40

11 Bibliography ............................................................................................................................. 41

List of Tables
Table 1 Size of generated data ........................................................................................................ 12
Table 2 Indexed columns in RDBMS ............................................................................................ 13
Table 3 Query 1 benchmark results .............................................................................................. 30
Table 4 Query 2 benchmark results .............................................................................................. 32
Table 5 Query 3 benchmark results .............................................................................................. 33
Table 6 Ignite cache loading time ................................................................................................. 33
Table 7 Apache Ignite release dates ............................................................................................. 34
Table 8 Product Comparison - Apache Ignite vs other in-memory distributed databases ............ 39

List of Figures
Figure 1 Ignite server and client node topology ........................................................................... 6
Figure 2 Partitioned cache mode. Shards are distributed among nodes, one node is the primary
holder of the data while the rest contain backups ........................................................................ 7
Figure 3 Replicated cache mode. All nodes possess all shards ..................................................... 8
Figure 4 Collocated joins (left) vs non-collocated joins (right) ...................................................... 8
Figure 5 ERD Schema ................................................................................................................... 11
Figure 6 Node startup from command line .................................................................................. 13
Figure 7 Node startup command line output ............................................................................... 14
Figure 8 Code startup ................................................................................................................... 14
Figure 9 Spring XML startup ....................................................................................................... 14
Figure 10 Ignite Web Agent system diagram ............................................................................. 15
Figure 11 Starting ignite-web-agent ............................................................................................ 15
Figure 12 Ignite Web Console ..................................................................................................... 16
Figure 13 Remote node discovery methods .................................................................................. 16
Figure 14 Add cluster .................................................................................................................... 17
Figure 15 Adding the JDBC driver ............................................................................................ 17
Figure 16 Import domain model screen ...................................................................................... 18
Figure 17 Setting connection properties .................................................................................... 18
Figure 18 Schema selection ......................................................................................................... 19
Figure 19 Table selection ............................................................................................................. 19
Figure 20 Additional configuration settings ................................................................................. 20
Figure 21 Imported domain model ............................................................................................. 20
Figure 22 Cache configuration as Java code ................................................................................. 21
Figure 23 Cache persistence configuration .................................................................................. 21
Figure 24 Generated code in a Maven project ............................................................................ 22
Figure 25 Configuration summary ............................................................................................... 22
Figure 26 Console output on running ServerNodeCodeStartup .................................................. 23
Figure 27 Loading the caches .................................................................................................. 24
Figure 28 Query 1 benchmark results (chart) .......................................................................... 31
Figure 29 Ignite cache loading (chart) ..................................................................................... 33
Figure 30 GridGain Pricing ...................................................................................................... 35
Figure 31 GitHub Pulse Report on the Apache Ignite repository [15] ...................................... 35
Figure 32 Contributions to master since Ignite open-sourcing in 2014 [16] ............................. 36
Figure 33 Commit frequency, year-to-date [17] ...................................................................... 36
Figure 34 Most active contributors [16] ................................................................................ 36
Figure 35 Searching for "ignite" in StackOverflow .................................................................. 37
Figure 36 Web Console Queries page .................................................................................... 38
Figure 37 Querying the Ignite cache from DBeaver ................................................................. 38
Figure 38 GridGain vs Hazelcast Performance Summary ....................................................... 39
1 Introduction

Acme Corp. is a fictitious company in the business of equipment rental to individuals as well as businesses. The equipment they lease are electronic devices such as laptops, desktop computers, projectors, multifunctional printers, monitors, tablets, servers, electronic blackboards, videoconferencing systems, lab equipment and smartphones. At present, the company has more than 3 million of these “assets” under management, and regular intervals the company needs to generate documents and reports out of this dataset. They are therefore exploring recent technologies like distributed processing and in-memory databases to improve their existing operations, to scale up their business and to explore other profitable ventures by extracting new business ideas from the data they already have.

To this end the company has engaged a team of data analytics specialists to study and recommend the most suitable technologies to use before they invest in new hardware, software and hiring. The company is asking for a proof of concept with benchmarks in order to compare their database management system (Microsoft SQL Server 2016) with an alternative solution that offers in-memory and distributed processing. In this paper we explore Apache Ignite (version 2.2) as an in-memory computing platform.

2 Apache Ignite

Apache Ignite is an open source in-memory distributed database and computing platform. It was initially released in 2007 by GridGain Systems (Foster City, California), open-sourced in 2014 and graduated from the Apache Incubator program in September 2015.

In this section we discuss the core features of Apache Ignite. More details can be found in the Apache Ignite documentation [1].

2.1 Clustering

There are two types of cluster nodes in Ignite: server or client.

- **Server** nodes act as containers for data, processing of aggregation requests. These build up the distributed database. The more server nodes, more RAM and CPU is available for the workload.
- **Client** nodes are entry points from applications. They are embedded in application logic, they function as a gateway to the cluster that is composed of the server nodes.
Node Discovery

Ignite provides several node discovery options, such as Multicast IP or Static IP discovery, or both. Apart from Java or XML configuration, the filesystem can also be used to store the nodes’ IP addresses. Similarly, node discovery can be configured in any of the cloud services that support Ignite.

Cluster Deployment Options

Ignite has flexible deployment options: it can be deployed on-premise or on-cloud, on physical servers or virtual environments. Ignite can be deployed from Docker, Kubernetes or Mesos containers. Additionally, images are available in both AWS (ignite-ami) and Google Compute (ignite-image) for quickly deploying Ignite clusters on the cloud.

Other Features

The following list summarizes other features relating to Ignite’s clustering function:

- Cluster grouping: ability to create logical groups of nodes within the cluster, which provides the ability to assign specific jobs or tasks to a subset of nodes only.
- Leader election: ability to select oldest or youngest nodes in a cluster, in situations a coordinator node is needed for certain tasks.
- Peer classloading: a special distributed ClassLoader provides zero deployment by avoiding explicit re-deployment of code to nodes every time it changes.

2.2 Durable Memory and Persistence

Ignite’s durable memory architecture allows storing and processing of data and indexes both on memory and disk, similar to virtual memory of operating systems. Persistence is optional: Ignite can be used as a pure in-memory store. However, built-in persistence called Ignite Native Persistence is provided for writing data to disk, and transparently integrates with Ignite’s durable memory.
2.3 Data Grid

In Apache Ignite’s distributed storage, each node owns a portion of the overall data. It stores data as key-value pairs in a distributed partitioned hashmap, stored in-memory. The data grid implements the JCache (JSR 107) specification which provides support for basic cache operations, ConcurrentMap APIs, collocated processing, events and metrics, etc.

Cache Modes

Ignite supports three Cache Modes: partitioned, replicated or local cache.

- **PARTITIONED**: the most scalable distributed cache mode, and is the default cache mode. The data set is divided into partitions (sharding) and all partitions are split equally between participating nodes. The number of backup nodes can be configured for each cache. Updates are applied to the primary node and the change is propagated at some point to the backup nodes. Update performance is good because only one node needs to be updated, but read performance may suffer because some nodes may have expired copies of the data. However, this backup behavior can be configured to be fully synchronous.

- **REPLICATED**: in this mode, all data is replicated in all nodes in the cluster. Good read performance but updates are expensive because all changes need to be propagated to all cache nodes.
INFOH415 – In-Memory databases with Apache Ignite

- **LOCAL**: this is the most lightweight cache mode as no data is distributed to the other nodes. This is ideal for read-only data or data that is only refreshed at a set frequency.

**Distributed Joins**

Ignite provides the ability to collocate compute with data or data with data to improve performance. This is called affinity collocation. Cache key objects can be annotated with `@AffinityKeyMapped` to mark the relationships to other objects that should be located in the same node. If affinity keys are not set and distributed joins are not enabled, join results may not be complete because non-collocated joins are disabled by default.

In distributed collocated joins, the join operation is performed locally per node and aggregated at the client side. Because data is only joined locally, it is possible that only partial results are returned if the corresponding key is not present on the same node. On the other hand, in distributed non-collocated joins, each node will send broadcast requests to other nodes in the cluster to retrieve the missing data. This is a more expensive join operation because it involves additional broadcast messaging and data movement. Please see Figure 4 for an illustration of this concept.

---

**Figure 3** Replicated cache mode. All nodes possess all shards

**Figure 4** Collocated joins (left) vs non-collocated joins (right)
Cache Atomicity Modes

There are two cache atomicity modes in Ignite: atomic or transactional. In **ATOMIC** mode, atomicity and consistency is guaranteed for a single operation. In **TRANSACTIONAL** mode it is possible to group several operations into one logical grouping that cannot be interleaved and guarantees ACID compliance.

Cache Querying

There are many ways to implement a cache query in Ignite, such as scan queries, SQL queries or text queries.

- **ScanQuery** provides a way of querying the distributed cache according to a predicate in full Java code. It can be in lambda form (Java 8) or anonymous functions (Java 7).
  ```java
  try {QueryCursor cursor = cache.query(new ScanQuery<Integer, ContractItem>((k, p) -> p.getPricePerDay().compareTo(Decimal.ONE)) { 
      for (Object p : cursor) { 
          System.out.println(p.toString()); 
      } 
  }
  ```

- **SQLQuery** provides a way to query using an SQL predicate.
  ```java
  SQLQuery query = new SqlQuery(Invoice.class, IQ: "contract_id = ? AND billing_date = ?");
  QueryCursor<List<?>> cursor = invoiceCache.query(query.setArgs(computeItem.getContractId(), billingDate));
  for (List<?> row : cursor) {
      invoiceId = (Integer) row.get(0);
      break;
  }
  ```

- **SQLFieldsQuery** provides a way to query only specific fields from a distributed cache.
  ```java
  SQLFieldsQuery sql = new SqlFieldsQuery("SELECT i.invoice_id FROM Invoice i WHERE i.contract_id = ? AND i.billing_date = ?");
  QueryCursor<List<?>> cursor = invoiceCache.query(sql.setArgs(computeItem.getContractId(), billingDate));
  for (List<?> row : cursor) {
      invoiceId = (Integer) row.get(0);
      break;
  }
  ```

- **SQLTextQuery** provides a way to do a text search in any column in a cache.
  ```java
  TextQuery txtquery = new TextQuery(Item.class, txt "Macbook");
  QueryCursor<Cache.Entry<Integer, Item>> query = itemCache.query(txtquery);
  for (Cache.Entry<Integer, Item> integerItemEntry : query) {
      System.out.println(integerItemEntry.getValue().toString());
  }
  ```

Other Features

The following list summarizes other features relating to Ignite’s data grid function:

- Near Cache: smaller, local caches on client nodes that stores most recently or frequently accessed data
- Cache grouping: merge caches into groups to reduce overhead and improve performance
- Pessimistic locks: enforce mutual exclusion through explicit locking
- Continuous queries: continuously query real-time listening of data modifications on Ignite caches
• Data rebalancing: Configurable automatic rebalancing of shards across nodes in a cluster in response to changes in topology

2.4 Distributed SQL
Apache Ignite is a ANSI-99 compliant horizontally scalable and fault-tolerant distributed SQL database, by full replication or partitioning. It supports both collocated and non-collocated distributed SQL joins as already described in a previous section. It supports both DDL (Data Definition Language), for example table creation), and DML (Data Manipulation Language) such as queries. Ignite ODBC and JDBC drivers are available so you can use your SQL tool of choice, or establish a connection from your source code.

The following list summarizes other features relating to Ignite’s distributed SQL function:

• Support for geospatial data (OGS Simple Features Specification)
• Ability to connect Tableau, Pentaho (data visualization tools) and Apache Zeppelin (data analytics notebook tool) to analyze data stored in a distributed Ignite cluster

2.5 Compute Grid features
Ignite provides distributed parallel processing: computations and data processing are spread across a set of nodes in the cluster. As already discussed in a previous section, it provides the ability to run computation on the node where the data is to avoid data serialization.

The following list summarizes other features relating to Ignite’s compute grid function:

• In-memory MapReduce: Run MapReduce and ForkJoin jobs in memory
• Continuous mapping: Ability to generate new jobs on-the-fly for the Map step when computation is already running
• Shared node state: Ability to share state between different jobs in a node
• Fault tolerance: Configurable job failover in case of node crash
• Load balancing: Configurable job distribution among cluster nodes
• Checkpointing: Ability to save an intermediate job state to protect from node failures
• Job Scheduling: Fine-grained control over scheduling of jobs that arrive at a node

2.6 Other interesting features
The following list summarizes other interesting features or integrations available in Ignite:

• Data streaming: ability to inject large amounts of continuous streams of data into Ignite caches. Provides integration with major streaming technologies and frameworks such as Kafka, Camel, Storm, Flume, Flink, MQ, among others
• Hibernate L2 cache: ability to be used as Hibernate’s second-level cache, caching of retrieved data to avoid expensive database operations
• Machine Learning grid: run machine learning algorithms on data stored in Ignite caches and avoid having to ETL data out into another system like Mahout or Spark
3 Business domain

As mentioned above, Acme Corp. is in the business of leasing high value equipment to individuals and businesses, who prefer to pay a weekly or monthly rate for the use of these assets in exchange for convenience and flexibility as they will not need to maintain these assets, and to avoid the large up-front cost of acquiring them. To start a lease a client fills out an information sheet and signs a contract that indicates the terms of the lease, which include the start and end dates of the lease, the rental rate, terms of rental rate adjustments, the billing interval, any charges for pre-terminating the contract, charges for damages, etc. There are five types of billing periods: Weekly, monthly, quarterly, semiannual and annual. Each client chooses the option that best fits their necessity.

At regular intervals, Acme Corp. generates an invoice by executing a batch job. The user logs into the application, selects parameters and clicks on a button that will generate the invoices in PDF format. These invoices are then sent out electronically to all clients.

Apart from this, the company also has other operations such as bulk insert and update of records, and report generation. Due to the volume of data involved these jobs are typically ran overnight. This is also a potential area for improvement if the in-memory computing solution shows significance performance improvements versus a traditional database. We will also cover some of these queries in this analysis.

4 Database schema and data setup

A subset of Acme Corp’s existing database schema can be seen in the Figure 5.

We can see a mapping to the business domain in the schema. A Customer has a first name, a last name and a billing address. The customer is associated with zero or many Contracts. These contracts have a start date, an end date and a billing interval (as mentioned before, weekly, monthly, quarterly, semiannual or annual). The contract contains one or many contract items. ContractItem is a table for storing the rental rate of the items. It contains a price per day, and a discount rate. Every item owned by Acme Corp is stored in the table ItemInstance. In this table we store the serial number of the item, the purchase date, the purchase amount, and its condition (working or defective). Each type
of item is stored in the table Item, and it contains the name of the item (this is, the commercial name of the product, for example “Macbook”), the brand, the type (as mentioned before, laptops, desktop computers, projectors, multifunctional printers, monitors, tablets, servers, electronic blackboards, videoconferencing systems, lab equipment and smartphones). We also store the manufacturer and a short description of the item. Each item instance has an accessory, stored in the table Accessory, with a name.

Everything related with the billing is stored in two tables: InvoiceItem and Invoice. When we want to calculate an invoice for a date and a specific billing period, we invoke a stored procedure that calculates the amount for each item for each contract match the parameters. This is stored in InvoiceItem (base amount, discount amount and net amount). Finally, the Invoice aggregates the InvoiceItems computed for that billing period and contract.

For this this POC we have created a test database with randomly generated data. In our dummy database we can find the following tables and their corresponding volume:

<table>
<thead>
<tr>
<th>Table</th>
<th># of records</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer</td>
<td>1,000,000</td>
</tr>
<tr>
<td>Contract</td>
<td>1,200,000</td>
</tr>
<tr>
<td>ContractItem</td>
<td>3,120,000</td>
</tr>
<tr>
<td>ItemInstance</td>
<td>3,120,000</td>
</tr>
<tr>
<td>Accessory</td>
<td>3,120,000</td>
</tr>
<tr>
<td>Item</td>
<td>1,000,000</td>
</tr>
<tr>
<td>Invoice</td>
<td>0</td>
</tr>
<tr>
<td>InvoiceItem</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1 Size of generated data

1 million customers have 1.12 million Contracts associated (the first 20,000 customers have two active contracts, and the next 100,000 customers have one inactive contract and one active contract), 3.12 million ContractItems, 3.12 million ItemInstances (3 million working items, 120,000 defective items) with 3.12 million Accessories and 1 million Items. Tables Invoice and InvoiceItem are empty. As we said, this data is randomly generated by a script. The contracts are uniformly distributed among the different billing_intervals.

As we can see, the main job of this database is to extract, compute and load invoices. No new records will be inserted on tables Customer, Contract, ContractItem, ItemInstance, Accessory or Item.

Finally, below in Table 2 is a summary of the indexed columns, either from primary keys or index definitions.

<table>
<thead>
<tr>
<th>Table</th>
<th>Indexed column</th>
<th>By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer</td>
<td>customer_id</td>
<td>Primary key (clustered)</td>
</tr>
<tr>
<td>Contract</td>
<td>contract_id</td>
<td>Primary key (clustered)</td>
</tr>
<tr>
<td>Contract</td>
<td>customer_id</td>
<td></td>
</tr>
<tr>
<td>Contract</td>
<td>start_date, end_date, billing_interval</td>
<td></td>
</tr>
<tr>
<td>Accessory</td>
<td>accessory_id</td>
<td>Primary key (clustered)</td>
</tr>
<tr>
<td>Accessory</td>
<td>item_instance_id</td>
<td></td>
</tr>
<tr>
<td>ContractItem</td>
<td>contract_item_id</td>
<td>Primary key (clustered)</td>
</tr>
<tr>
<td>ContractItem</td>
<td>contract_id</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2 Indexed columns in RDBMS

<table>
<thead>
<tr>
<th>Table</th>
<th>Column</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ContractItem</td>
<td>item_instance_id</td>
<td></td>
</tr>
<tr>
<td>Item</td>
<td>item_id</td>
<td>Primary key</td>
</tr>
<tr>
<td>ItemInstance</td>
<td>item_instance_id</td>
<td>Primary key</td>
</tr>
<tr>
<td>ItemInstance</td>
<td>item_id</td>
<td></td>
</tr>
<tr>
<td>Invoice</td>
<td>invoice_id</td>
<td>Primary key</td>
</tr>
<tr>
<td>Invoice</td>
<td>contract_id, billing_date</td>
<td></td>
</tr>
<tr>
<td>InvoiceItem</td>
<td>contract_item_id, invoice_id</td>
<td>Primary key</td>
</tr>
</tbody>
</table>

5 **Apache Ignite Walkthrough**

#### 5.1 Environment Setup

Download and install `apache-ignite-fabric` from [https://ignite.apache.org/download.cgi](https://ignite.apache.org/download.cgi). Then add a `IGNITE_HOME` environment variable as well as to `PATH` (pointing to `IGNITE_HOME/bin`).

#### 5.2 Cluster/Node startup

There are two ways to start an Ignite node: through Java code, or through command line. If there is no existing cluster with the same node configuration, a new cluster will be created. Otherwise the new node will join the existing cluster.

##### 5.2.1 Via Command Line

To create a node, run `ignite.[bat|sh]` and provide the path of the XML configuration file. For example:

```
C:\Windows\System32\cmd.exe - ignite.bat config/acmecorp.xml
```

![Figure 6 Node startup from command line](image)

---

13
5.2.2 From Java

Nodes can be started by calling `Ignition.start()`. Similar to command line, you can pass the path of the XML configuration file, or use the Java configuration code. Both examples are shown below. These files are provided in the auto-generated Maven project created from the Web Console, described in section 5.3.
5.3 Domain Model generation

Once the database tables have been created, we can use Ignite’s Web Console [2] to automatically generate the domain model from the database schema. The domain model, in Java or XML code, can be downloaded as a Maven package and used in your application. The import is performed through a wizard-style form with a live preview of the configuration code.

Web Console can be deployed locally, but for convenience, GridGain also hosts an instance that is accessible for free on the internet through http://console.gridgain.com/.

To use the Web Console, one needs to download a web agent that establishes a connection between the Ignite cluster and the Web Console [3]:

![Figure 10 Ignite Web Agent system diagram](image)

The link to download `ignite-web-agent` is provided from the Web Console interface, already pre-configured with the appropriate security token. Unzip the package and run `ignite-web-agent.bat`. At least one cluster node running from `apache-ignite-fabric` (command line) should be started before running the agent.

![Figure 11 Starting ignite-web-agent](image)

If the connection is successful, Web Console will show the name of the cluster detected.
The first step is to configure an Ignite cluster. From the header, click on **Configure** and then the **Advanced** tab. Click on the **Add cluster** button and give a name for the cluster. We will use the default **Multicast** discovery method for finding remote nodes on the grid. Other available discovery methods can be found by hovering over the (?) icon.
As you modify some of these fields you will notice that the XML/Java code at the right side changes as you type or modify selections. This is a useful feature that shows you exactly what your changes will do to the configuration code, which you can use to look up in the Ignite API documentation or Javadoc for more information on their behavior.

If you want other nodes in a remote host to connect to the cluster, you will need to check that the firewall rules are not blocking the connection.\(^1\)

Save the changes in the **Clusters** page and go to the **Model** section from the left side menu.

To import the database schema, a JDBC driver must be provided in the **ignite-web-agent-*/jdbc-drivers** directory:

After doing so, click the **Import from database** button.

![Import domain model screen](image)

*Figure 16: Import domain model screen*

In the next screen specify the connection details to the database where the tables have been created.

![Setting connection properties](image)

*Figure 17: Setting connection properties*

In the next screen select the appropriate schema.
In Memory databases with Apache Ignite

In the next screen we select the tables to be imported. Select the appropriate tables. At the bottom there is an option to choose different Cache Modes: PARTITIONED or REPLICATED – leave the defaults for now and click Next.

Accept defaults and click Save.
You will see that a list of caches has been generated with default settings. It is possible to modify the configuration for each cache, such as setting the Cache Mode (PARTITIONED/REPLICATED/LOCAL) and Cache Atomicity (ATOMIC/TRANSACTIONAL). Please refer to section 2.3 for a description of these features.

Click on the Java label to view the configuration as Java code.
Scrolling down under the **Store** section there are more configurable settings on persistence.

Once satisfied with the cache configuration, go to the **Summary** section. You will see a preview of the configuration code based on selections made in the previous screens. The source code can be downloaded as a ready-to-use Maven project. You can view the files and directories created by clicking the **Project Structure** button.
Click on the **Download Project** button and open `pom.xml` from your favorite Java IDE.
Inside the `src/main/java/startup` directory there are startup classes which you can use to start server or client nodes, based on either XML or Java configuration files.

5.4 Loading caches

After initializing the node, the caches can be accessed through the `ignite.cache()` or `ignite.getOrCreateCache()` call. The only difference between these two is the `cache()` call will return null if the cache does not exist, while the `getOrCreateCache()` will create the cache [4].

Once you have a handle to the cache, you may load the data by calling the `loadCache()` method. This call reads from the tables in the database and loads them all into memory, automatically sharding the data across nodes according to the configuration given in the Java or XML code. It already knows the schema of your data when it auto-generated the domain model (as seen in section 5.3). The connection is established according to the credentials given in `secret.properties` (also included in the Maven project).
Figure 27: Loading the caches

5.5 Computing cache data

Once the application has a reference to the cache with data already loaded, it is possible to run any of the query types described in section 2.3. In section 6 we will discuss the different queries we implemented for this POC. For additional examples you may refer to the Ignite documentation [1] or the ignite-examples project [5].
6 Queries

In this section we explain the queries that we developed in both SQL and Apache Ignite. These queries are representative of the typical database operations that will be done in normal business processing.

6.1 Query 1: Multiple operations in a single procedure (generate invoices)

This job is a procedure that will generate invoices that will be sent out to clients regularly. As described in section 3, improving the performance of this job is our main goal in this proof of concept. The job can be broken down into the following:

Parameters:

- **billing_date** – date at which contracts are active to be picked up by the job
- **billing_interval** – billing interval of contracts to be picked up by the job

Steps:

1. Query the `Contract` and `ContractItems` tables to get a list of items that are billable (according to given parameters)
2. Delete any existing `InvoiceItem` records for the given parameters
3. Delete any existing `Invoice` records for the given parameters
4. Insert new `Invoice` records for each distinct contract in Step 1
5. Insert new `InvoiceItem` records for each `ContractItem` returned in the Step 1 and set its foreign key to `Invoice`, which are the values inserted in Step 4.

This procedure is roughly the same in both the SQL implementation and in Ignite, but could differ in that in SQL it is possible to do an `INSERT INTO SELECT` that bulk-computes and bulk-inserts rows into the table, whereas in Ignite we have to loop over each one.

6.1.1 SQL

Below is the code for generating invoices along the same steps described in the beginning of section 6.1.

Step 1: Query billable items

```sql
--Retrieve items to be computed
INSERT INTO @ContractItemsToCompute
SELECT
    c.contract_id,
    ci.contract_item_id,
    ci.price_per_day,
    ci.discount_rate
FROM contract c,
     ContractItem ci
WHERE c.contract_id = ci.contract_id
    AND c.billing_interval = @billingInterval
    AND c.start_date <= @billingDate
    AND (c.end_date = '31-DEC-9999' OR @billingDate < dateadd(DAY, 1, c.end_date));
```

Step 2: Delete existing `InvoiceItem` records
DELETE FROM InvoiceItem;

Step 3: Delete existing Invoice records

DELETE FROM Invoice;

Step 4: Insert new Invoice records

\--- from page 0 to page N times, buckets of rangeSize, Starting from @from tuple
WHILE @pos <= @numberOfInvoicePages
BEGIN
BEGIN TRANSACTION;
INSERT INTO Invoice (contract_id, billing_date, total_amount)
SELECT T.contract_id, @billingDate, 0
FROM (SELECT DISTINCT cic.contract_id AS contract_id
FROM @ContractItemsToCompute cic
ORDER BY contract_id ASC
OFFSET @from ROWS
FETCH NEXT @rangeSize ROWS ONLY) T

--move to next page
SET @pos = @pos + 1;

--set the next "first tuple" to the last page's tuple
SET @from = @from + @rangeSize
COMMIT TRANSACTION;
END

Step 5: Insert new InvoiceItem records

WHILE @pos <= @numberOfInvoicePages
BEGIN
BEGIN TRANSACTION;
INSERT INTO InvoiceItem (invoice_id, contract_item_id, base_amount, discount_amount, net_amount)
SELECT
(SELECT i.invoice_id
FROM Invoice i
WHERE i.contract_id = c.contract_id AND i.billing_date = @billingDate ORDER BY i.invoice_id ASC
OFFSET @from ROWS
FETCH NEXT @rangeSize ROWS ONLY),
c.contract_item_id,
@chargeDays * c.price_per_day,
@chargeDays * c.price_per_day * (c.discount_rate / 100),
((@chargeDays * c.price_per_day) -
(ChargeDays * c.price_per_day * (c.discount_rate / 100)))
FROM @ContractItemsToCompute c;

--move to next page
SET @pos = @pos + 1;

--set the next "first tuple" to the last page's tuple
SET @from = @from + @rangeSize
COMMIT TRANSACTION;
END
6.1.2 Apache Ignite

Below is the code for generating invoices along the same steps described in the beginning of section 6.1.

Step 1: Query billable items

```java
try (Transaction tx = Ignition.ignite(NAME: "acemcorp").transactions().txStart()) {
    try (QueryCursor<billableItem> cursor = getComputeCursor(contractCache, billingIntervalType, billingDate)) {
        for (billableItem : cursor) {
            computeItems.add(new ComputePOJO(row));
        }
    }
}
```

The main things of interest here is the transaction boundaries around the `getComputeCursor` call (the `try()` and `tx.commit()` statements). In the `getComputeCursor` call we can see that the query is SQL-likem apart from the second line where we prefix the name of the `ContractItemCache` to the `ContractItem` table. This is because the query is run against `contractCache` but it has a reference to another cache. Also, note that we call `setDistributedJoins` to `true`. This is to ensure that we get complete results – this behavior has been described in section 2.3 under distributed joins.

Step 2: Delete existing InvoiceItem records

```java
try (Transaction tx = Ignition.ignite(NAME: "acemcorp").transactions().txStart()) {
    SqlFieldsQuery sql = new SqlFieldsQuery("DELETE FROM InvoiceItem");
    invoiceItemCache.query(sql);
    tx.commit();
    System.out.println("[x] Cleared InvoiceItem.xm", stopwatch.toSplitString());
}
```

This is similar to the previous query but a different SQL statement.

Step 3: Delete existing Invoice records

```java
try (Transaction tx = Ignition.ignite(NAME: "acemcorp").transactions().txStart()) {
    SqlFieldsQuery sql = new SqlFieldsQuery("DELETE FROM Invoice");
    invoiceCache.query(sql);
    tx.commit();
    stopwatch.split();
    System.out.println("[x] Cleared Invoice.xm", stopwatch.toSplitString());
}
```

This is nearly identical to the query in Step 3, but against the `Invoice` table.
Step 4: Insert new Invoice records

```java
try (Transaction tx = Ignition.ignite(name: "acmecorp").transactions().txStart()) {
    generateInvoices(contractIds, billingDate);
    tx.commit();
    stopwatch.split();
    System.out.println("[X] Generated Invoices...Xn", stopwatch.toString());
}
```

This operation is the code that generates the Invoice records. Some of these lines have been discussed in previous steps. The thing to note here is the `invoiceCache.put` call – this is the act of writing to the cache. As you can see we don’t need to write explicit SQL code – Ignite will manage the persistence to SQL database.

Step 5: Insert new InvoiceItem records

```java
private void generateInvoiceItems(List<ComputerPOD> computeItems, BillingIntervalType billingIntervalType, Date billingDate) {
    int i = 0;
    for (ComputerPOD computeItem : computeItems) {
        InvoiceItem invoiceItem = computerItem.calculateInvoiceItem(billingIntervalType, computerItem.getContractId(), billingDate);
        Integer invoiceId = 0;
        sqlFields = new SQLFieldsQuery("SELECT 1.invoice_id FROM Invoice I WHERE I.contract_id = ? AND I.billing_date = ?")
            .setDistributedJoins(true);
        QueryCursor<SqlFieldsQuery> queryCursor = invoiceCache.query(sql.setArgs(computerItem.getContractId(), billingDate));
        for (i < queryCursor.size) cursor " invoiceCache.get(sql) {
                invoiceId = (Integer) row.get(0);
                break;
            }
        invoiceItemKey = new InvoiceItemKey(invoiceId, computerItem.getContractId());
        invoiceCache.put(invoiceItemKey, invoiceItem);
        i++;
    }
}
```

This query does the work of the actual calculation of the InvoiceItem. The syntax is similar to what is already seen in previous steps.
6.2 Query 2: Aggregate by sum according to a date range (compute revenue)

In this example we will see how the revenue is computed in two ways, with SQL and with Apache Ignite. We can calculate the total revenue in two ways. First one is to obtain all Invoices from a specific period, and then, make a SUM aggregation in order to obtain the revenue. The other way is by Joining Contracts, Invoices and InvoiceItems, and then, making the SUM aggregation in the InvoiceItems, using the Invoice table as a bridge to obtain the billing period from the contract. As we can see, this last option is more complex and it has approximately three times more data in our database. Since we want to test our system, we decided to proceed with this last method.

6.2.1 SQL

In contrast with the previous query, we do not need to create a stored procedure.

```sql
SELECT c.billing_interval AS INTERVAL, SUM(ii.net_amount) AS TOTAL
FROM Invoice i, Contract c, InvoiceItem ii
WHERE c.contract_id = i.contract_id AND ii.invoice_id = i.invoice_id
AND i.billing_date BETWEEN '1-DEC-2017' AND '31-DEC-2017'
GROUP BY c.billing_interval
```

As we can see, we project the billing interval and the sum of the individual net amounts of each contractItem for the period between 1st of December 2017 and 31st of December 2017.

6.2.2 Apache Ignite

For querying the Ignite cache, we are going to use the Web Console application. As we can see in the figure below, the syntax is very close to SQL. The only appreciable difference is that we need to specify the cache name for Invoice and InvoiceItem in order to obtain the tables, because they reside in a different cache.

The output result of the query can be found below.
6.3 Query 3: Update all rows of a single column (apply discount)
In this query the aim is to explore the performance of an `UPDATE` statement on the entire table with 3,120,000 rows. We will update the entire `ContractItem` table and add a 5% discount to all items.

6.3.1 SQL

```
UPDATE ContractItem SET discount_rate = discount_rate + 5;
```

6.3.2 Apache Ignite
Since the query involves only a single cache, the same SQL can be executed as-is in Ignite’s Web Console application.

7 Benchmark results
In the following section we summarize the benchmark results.

7.1 Query 1 result
The benchmark result of Query 1 is summarized in the table below.

<table>
<thead>
<tr>
<th>ID</th>
<th>Ran on</th>
<th>Topology</th>
<th>Time</th>
<th>Total rows inserted</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ignite cluster</td>
<td>1 local node</td>
<td>0 m 50.4 s</td>
<td>292,780</td>
</tr>
<tr>
<td>2</td>
<td>Ignite cluster</td>
<td>2 local nodes</td>
<td>1 m 47.6 s</td>
<td>292,780</td>
</tr>
<tr>
<td>3</td>
<td>Ignite cluster</td>
<td>3 local nodes</td>
<td>2 m 44.7 s</td>
<td>292,780</td>
</tr>
<tr>
<td>4</td>
<td>Ignite cluster</td>
<td>4 local nodes</td>
<td>7 m 05.3 s</td>
<td>292,780</td>
</tr>
<tr>
<td>5</td>
<td>Ignite cluster</td>
<td>3 local nodes, 1 remote node</td>
<td>19 m 40.4 s</td>
<td>292,780</td>
</tr>
<tr>
<td>6</td>
<td>Ignite cluster</td>
<td>3 local nodes, 2 remote nodes</td>
<td>23 m 08.7 s</td>
<td>292,780</td>
</tr>
<tr>
<td>7</td>
<td>SQL Server Stored Procedure</td>
<td>n/a</td>
<td>0 m 09.0 s</td>
<td>292,780</td>
</tr>
</tbody>
</table>

*Table 3 Query 1 benchmark results*

We ran Query 1 using a mix of topologies: some with remote nodes, some without, to compare the performance as we modify the topology of the cluster.
1 Local node vs Stored Procedure comparison

As we can see, the best performance is seen on the SQL Server stored procedure, which ran for 9 seconds versus Ignite’s one-node execution (50.4 seconds) – see results {1, 7} in Table 3. We believe it is due to the following reasons:

- The stored procedure was using an `INSERT INTO SELECT` query which just copies data from one table to another with a simple computation inside (multiplication). On the other hand, the Ignite implementation requires the use of cursors and looping over all results individually, doing the calculation, and writing the result back.
- Second, the stored procedure is executing on the same database server as the data. On the other hand, although the Ignite implementation is ran on the same machine as the SQL Server database, we believe that the overhead in loading the data into Ignite’s in-memory data structure is a contributing factor to its slower performance.
- Third, SQL Server caches the query plan the first time a query is executed. On the other hand, ad-hoc queries can also be cached but may be evicted due to memory usage [6]. The procedure was not defined with the `WITH RECOMPILE` option, which means that the stored procedure’s query plan was cached. [7]
- Fourth, using Ignite involves overhead that do not apply to the stored procedure. For example, data in Ignite caches all need to be `SERIALIZABLE` for transmission between nodes. This marshalling-unmarshalling of data is just one of the overhead operations that do not apply to the stored procedure implementation.
- Fifth, Microsoft SQL Server is a commercial database product in existence since 1989. Over the decades it has been optimized to perform well in scenarios like this. In contrast, Apache Ignite was launched in 2015 and could probably still be optimized to perform better.
Local node only comparisons

Inspecting the results among the local Apache Ignite executions (see results {1, 2, 3, 4} in Table 3), we can observe that the execution time rises proportionally with the number of nodes in the cluster. We believe this is due to the following reason:

- The implementation of the domain objects is not collocated. We tried to implement affinity keys but it requires an object key, whereas the generated key from Web Console is a simple Integer, which does not satisfy the requirement for defining affinity keys. [8] This means that any join performed requires multiple broadcast messages to retrieve the missing data, as described in section 2.3 under Distributed Joins.

Local node and remote node comparisons

Lastly, inspecting the results between local nodes and remote nodes (see results {3, 4, 5, 6} in Table 3), we can see that adding a remote node increases the run time. We believe it is due to the following reasons:

- Similar to the previous analysis, the main culprit is non-collocated joins.
- The first reason is worsened by the fact that the broadcast and data transfers are happening over a network.
- The network over which the cluster nodes were connected is a Wi-Fi network. It is known that Wi-Fi networks have higher latency and interference versus wired Ethernet. [9] We would be interested to retest the benchmark over a wired network.

7.2 Query 2 result

The benchmark result of Query 2 is summarized in the table below.

<table>
<thead>
<tr>
<th>ID</th>
<th>Ran on</th>
<th>Topology</th>
<th>Time</th>
<th>Total rows selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ignite cluster</td>
<td>3 local nodes</td>
<td>0m 05.9 s</td>
<td>Inner join of 3 tables with (1120000 x 1208713 x 408193) rows</td>
</tr>
<tr>
<td>2</td>
<td>SQL Server Query</td>
<td>n/a</td>
<td>0m 02.3 s</td>
<td>Inner join of 3 tables with (1120000 x 1208713 x 408193) rows</td>
</tr>
</tbody>
</table>

Table 4 Query 2 benchmark results

Here we find a similar scenario than in Query 1 wherein SQL Server is still faster than Apache Ignite. This is due to the same results as in Query 1, namely: affinity keys are not defined; non-collocation of query joins; serialized data for inter-node transmission.
7.3 Query 3 result

The benchmark result of Query 3 is summarized in the table below.

<table>
<thead>
<tr>
<th>ID</th>
<th>Ran on</th>
<th>Topology</th>
<th>Time</th>
<th>Total rows updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ignite cluster</td>
<td>2 local nodes</td>
<td>9 m 45.0 s</td>
<td>3,120,000</td>
</tr>
<tr>
<td>2</td>
<td>SQL</td>
<td>n/a</td>
<td>0 m 25.4 s</td>
<td>3,120,000</td>
</tr>
</tbody>
</table>

*Table 5 Query 3 benchmark results*

Query 3 is a single table operation that updates a single column. We expect that the Ignite update operation will take more time because it needs to update the cache in addition to the database update. However, at almost 10 minutes in Ignite vs. 25 seconds in SQL Server, the time difference is significant. We were not expecting the time difference to be this severe because the Ignite nodes were in a local cluster, and there is no table join in this query. Because it is difficult to interpret these results, we will need to run more tests to see if this behavior is consistent.

7.4 Additional: Ignite cache loading

As a side note we are listing some run times for loading the entire database into memory. Please refer to Table 1 in section 0 for the amount of records in the database.

<table>
<thead>
<tr>
<th>ID</th>
<th>Ran on</th>
<th>Topology</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ignite cluster</td>
<td>1 local node</td>
<td>3 m 04.0 s</td>
</tr>
<tr>
<td>2</td>
<td>Ignite cluster</td>
<td>2 local nodes</td>
<td>5 m 47.0 s</td>
</tr>
<tr>
<td>3</td>
<td>Ignite cluster</td>
<td>3 local nodes</td>
<td>9 m 03.0 s</td>
</tr>
<tr>
<td>4</td>
<td>Ignite cluster</td>
<td>4 local nodes</td>
<td>12 m 10.0 s</td>
</tr>
<tr>
<td>5</td>
<td>Ignite cluster</td>
<td>3 local nodes, 1 remote node</td>
<td>10 m 54.0 s</td>
</tr>
</tbody>
</table>

*Table 6 Ignite cache loading time*

We definitely see a correlation between the number of nodes and the time it takes to load the cache. We can observe this in runs {1, 2, 3, 4}. We can attribute this to load balancing of partitions/shards across nodes in the cluster.
However, the load time of (5), where the topology includes 1 remote node, is not as severe as what we saw in the benchmark results of Query 1 in section 7.1. We cannot draw a conclusion from this behavior as yet; we will need to execute more tests to verify whether this is an intermittent occurrence.

8 Development experience

As with any technology, the development experience is also an important consideration in a decision to adopt any library. One should choose a library that not only satisfies your technical requirement, but also aligns with other factors such as maintenance activity, ease-of-use, availability of developer skills, etc. In this section we briefly go through these factors and give a rating in this aspect of technology selection.

8.1 Open source vs closed source

In choosing a technology we must consider the how our dependency on the library will play out over the long term. In closed-source projects one has to pay licensing fees, and there is a risk that the company will end development and/or support to the product, which could be troublesome as it could necessitate significant development work to eliminate the dependency when those events happen. Therefore it is often a good idea to use open source libraries.

As mentioned in section 2, Ignite became an official Apache open source project since 2015. The latest version (2.3.0) was released in 31 October 2017, one and a half months after the previous release (2.2.0 in mid-September). Below is a summary of the last ten Ignite releases and their rc1 release dates [10].

<table>
<thead>
<tr>
<th>Version</th>
<th>Release Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.0 (rc1)</td>
<td>2017 Oct 27</td>
</tr>
<tr>
<td>2.2.0 (rc1)</td>
<td>2017 Sep 11</td>
</tr>
<tr>
<td>2.1.0 (rc1)</td>
<td>2017 Jul 13</td>
</tr>
<tr>
<td>2.0.0 (rc1)</td>
<td>2017 Apr 28</td>
</tr>
<tr>
<td>1.9.0 (rc1)</td>
<td>2017 Feb 28</td>
</tr>
<tr>
<td>1.8.0 (rc1)</td>
<td>2016 Dec 1</td>
</tr>
<tr>
<td>1.7.0 (rc1)</td>
<td>2016 Aug 1</td>
</tr>
<tr>
<td>1.6.0 (rc1)</td>
<td>2016 May 18</td>
</tr>
<tr>
<td>1.5.0 (rc1)</td>
<td>2015 Dec 1</td>
</tr>
<tr>
<td>1.4.0 (rc1)</td>
<td>2015 Sep 24</td>
</tr>
</tbody>
</table>

Table 7 Apache Ignite release dates

As can be seen here, Apache Ignite releases a new version approximately every 2-3 months. This frequency is at par with other “hot” libraries of today such as Apache Spark, or other popular in-memory databases like Memcached [11] or Redis [12].

As a side note, GridGain Systems, Ignite’s original developer, operates a business around Apache Ignite in two ways: selling enterprise-grade features or through support contracts. Below is a screenshot of GridGain’s pricing model [13].
8.2 Developer community

In choosing a library or a technology it is important to also assess the activeness of the developer community in implementing new features, upgrades or fixing issues. This can be measured through a few metrics such as the number of commits, size and demographics of the community, etc. [14]

The following figures show the available data on the Apache Ignite GitHub repository in https://github.com/apache/ignite.

**Figure 30 GridGain Pricing**

<table>
<thead>
<tr>
<th>Solution</th>
<th>Open Source</th>
<th>Standard Professional</th>
<th>Standard Enterprise</th>
<th>Premium Enterprise</th>
<th>Premium Ultimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintenance Releases</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>Hot Bug Fixes</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>Enterprise Features</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>Cluster Snapshots</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>Support Hours (Phone or Online)</td>
<td>9x5</td>
<td>9x5</td>
<td>24x7</td>
<td>24x7</td>
<td></td>
</tr>
<tr>
<td>SLA</td>
<td>1 Business Day</td>
<td>1 Business Day</td>
<td>1 Hour</td>
<td>1 Hour</td>
<td></td>
</tr>
<tr>
<td>Unlimited Support Incidents</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>Installation/Upgrade Assistance</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>Software indemnification</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>Price</td>
<td>Free</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 31 GitHub Pulse Report on the Apache Ignite repository [15]**
8.3 Developer support

Over the course of the development of our POC we had many questions that is not directly addressed in the documentation. In times like these it is crucial to have a network of other developers who can
respond to your questions, especially when working with a library that one has no prior experience in. For this we turn to the popular Q&A site StackOverflow. Depending on the question we usually receive a response within a few days. (Fun fact: the most frequent responder to our questions is a Lead Architect from GridGain systems)

There are 6,001 questions relating to “ignite” in StackOverflow, which is less than a similar search for other in-memory libraries such as Redis (47,128) or Memcached (24,073).

GridGain Systems also hosts frequent webinars on Apache Ignite that are both technical and non-technical on a range of topics, such as Ignite essentials, architecture, and specific use cases of in-memory computing platforms. We found these webinars to be very useful and provide a chance to ask Ignite questions to senior architects from GridGain. [18]

### 8.4 Documentation

The Apache Ignite documentation [1] is comprehensive and easy-to-read once you understand the basic concepts behind distributed data storage. Any configuration is explained with an accompanying code snippet in both Java and Spring XML. Additionally, the GitHub repository provides an `ignite-examples` subproject which one can use as reference for complete code examples.

### 8.5 Tooling

As shown in section 5.3, the Web Console is a very useful tool in understanding the configuration options behind an Ignite cluster and its caches. The live preview of form changes is particularly useful
if one already has an existing project but would like to configure a specific feature without regenerating the configuration files.

Additionally, the Queries section also works very well for ad-hoc querying of cache data. It provides the ability to query the entire cluster, or specific nodes, and supports both collocated and non-collocated joins.

![Figure 36 Web Console Queries page](image)

Apache Ignite also comes with a simple command-line tool called **Ignite Visor** which lets you do simple monitoring on the cluster, such as printing the cluster topology, node statistics, CPU and memory allocation, size of the caches; it also provides the ability to start or kill remote nodes.

We also tested using Ignite’s JDBC driver for querying cache contents. **DBBeaver** is a universal SQL client that connects to the caches through Ignite’s JDBC Driver. Results were spotty... sometimes it does not work (we get a parsing error on the SQL statement). Moreover, browsing the database from the Database Navigator is currently not yet supported.

![Figure 37 Querying the Ignite cache from DBBeaver](image)
As for the coding experience, writing lambda expressions for iterating over `ScanQuery` results has been challenging because errors only show up during runtime, not in compile time. However, this may be attributed to the developers’ relative inexperience with Java 8 lambda expressions in general.

9 Apache Ignite vs. other in-memory distributed databases

How does Apache Ignite stack up against other similar in-memory distributed databases? GridGain Systems has a few studies on this topic, as well as benchmark results, which are summarized in the sections below.

9.1 Feature comparisons

From GridGain’s product comparison page [19] we selected a few key features which are summarized in the table below.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Apache Ignite (GridGain)</th>
<th>Hazelcast</th>
<th>Oracle Coherence</th>
<th>Pivotal GemFire</th>
<th>GigaSpaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL Queries</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Continuous Queries</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Distributed SQL Joins</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Query Consistency</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Query Fault Tolerance</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>JDBC Driver</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>ODBC Driver</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Data Streamer</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Table 8 Product Comparison - Apache Ignite vs other in-memory distributed databases

As we can see from the above table, some core Ignite features is not even implemented in other in-memory databases. We believe this makes Apache Ignite a strong candidate to be used as a general purpose in-memory database.

9.2 Performance comparisons

The only available benchmark results on the GridGain website is against Hazelcast.

GridGain vs. Hazelcast Performance Summary

<table>
<thead>
<tr>
<th>Feature</th>
<th>GridGain</th>
<th>Hazelcast</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadlock Free Transactions</td>
<td>53% - 65% higher operations/sec throughput</td>
<td></td>
</tr>
<tr>
<td>Transactional Operations</td>
<td>30% - 57% higher operations/sec throughput</td>
<td></td>
</tr>
</tbody>
</table>
| Atomic Operations                | Put Benchmark: Up to 21% higher operations/sec throughput  
Put and Get Benchmark: Depending on configuration, either GridGain or Hazelcast performed slightly better |          |
| SQL Query                        | 13% - 40% higher operations/sec throughput |          |

Figure 38 GridGain vs Hazelcast Performance Summary
As can be seen in the figure above, GridGain asserts better performance over Hazelcast. For more information please refer to the GridGain vs Hazelcast benchmark page [20].

10 Conclusion

Based on this study we have the following conclusions.

First, the benchmark results for Query 1 showed that SQL stored procedures still have an edge over in-memory databases for this specific use case. However, we need to evaluate those results with a grain of salt, because the implementation in this study does not use affinity collocation (see section 7.1). Other observations stem from the benchmark results in adding local nodes and remote nodes. We can see that execution times are slower when more nodes are added, and significantly slower when remote nodes are added. Local-only clusters perform faster than clusters with remote nodes. This is due to several factors, as laid out in section 7. Given the hype around distributed processing in recent years, we should not always assume that it is a one-size-fits-all solution that works well in any use case. We think it is important to conduct a POC like this study to prototype key operations and check actual results before making a decision to adopt a technology.

Additionally, in-memory distributed storage requires configuration that may add complexity to an application. Each decision in the cluster and cache configuration needs to be assessed, because the optimal configuration or topology may vary depending on the size and read/write frequency of the data, as well as on compute job to be executed. Some jobs may require fast reads, and so a REPLICATED cache mode may be best; on the other hand, other jobs may require fast updates, where the PARTITIONED cache mode performs well. This needs to be balanced with the actual hardware resources available on the cluster. Apache Ignite provides sophisticated configuration options to fine-tune these settings but the design of the clusters and the caches need to be analyzed carefully to arrive at the optimal configuration that balances performance and cost.

We like that Apache Ignite is an open source library that is also backed by a company who can provide enterprise-grade features and support. In developing the prototype we appreciate the resources and documentation found online. However, this information is mostly concentrated in one or two sites or message boards, unlike more popular libraries where there are numerous blogs discussing the technology. Some GridGain engineers are actively responding to StackOverflow questions, however we wish that Apache Ignite would gain a larger following to increase the pool of developer support.
11 Bibliography


[10] "Apache Ignite Releas...


