GridGain

How to Migrate Your Data Schema to Apache Ignite

Ivan Rakov

December 4, 2019

How to Migrate Your Data Schema to Apache Ignite

December 4, 2019

Ivan Rakov

- Work at GridGain Systems
  - Leading data consistency dev team
- Apache Ignite Committer

GridGain

Agenda

- What is and what is not Ignite SQL: pros and cons

Agenda

- What is and what is not Ignite SQL: pros and cons
- Ignite SQL typical successful use cases

Agenda

- What is and what is not Ignite SQL: pros and cons
- Ignite SQL typical successful use cases
- How to cook Ignite SQL: four-step guide

Agenda

- What is and what is not Ignite SQL: pros and cons
- Ignite SQL typical successful use cases
- How to cook Ignite SQL: four-step guide
- Ignite SQL: performance tuning

Agenda

- What is and what is not Ignite SQL: pros and cons
- Ignite SQL typical successful use cases
- How to cook Ignite SQL: four-step guide
- Ignite SQL: performance tuning
- Living with Ignite SQL: schema evolution

Agenda

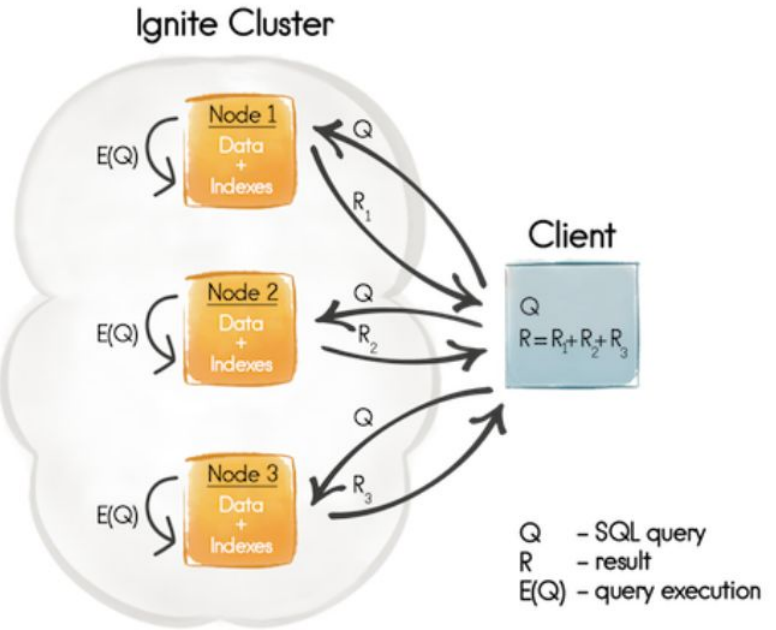
- What is and what is not Ignite SQL: pros and cons
- Ignite SQL typical successful use cases
- How to cook Ignite SQL: four-step guide
- Ignite SQL: performance fine-tuning
- Living with Ignite SQL: schema evolution

Ignite SQL

- Ignite can be used as distributed SQL database
  - ANSI-99 compliant
  - Horizontally scalable
  - Fault-tolerant

Ignite SQL

- Ignite can be used as distributed SQL database
  - ANSI-99 compliant
  - Horizontally scalable
  - Fault-tolerant
- Ignite SQL architecture
  - Tightly coupled with H2 database
    - parsing, optimizing, local query execution
  - Distributed logic based on map-reduce
  - Data is stored in Ignite Durable Memory
    - RAM offheap speed + optional disk durability



Ignite SQL is good for

- Providing SQL access to large datasets
  - not fitting well in one server

Ignite SQL is good for

- Providing SQL access to large datasets
  - not fitting well in one server
- Fault-tolerance
  - one node crashes, but cluster live and SQL works

Ignite SQL is good for

- Providing SQL access to large datasets
  - not fitting well in one server
- Fault-tolerance
  - one node crashes, but cluster live and SQL works
- Easy data distribution
  - data is partition according to affinity function, no need for manual sharding

Ignite SQL is good for

- Providing SQL access to large datasets
  - not fitting well in one server
- Fault-tolerance
  - one node crashes, but cluster live and SQL works
- Easy data distribution
  - data is partition according to affinity function, no need for manual sharding
- Providing better query performance than single database server
  - all cluster nodes work on your query simultaneously
  - **true when query fits well in a single map-reduce cycle**
    - **query fits in map-reduce cycle when data rows from different nodes don't interact**

Ignite SQL is not good for

- Consistent processing
  - Ignite SQL **is not transactional**

Ignite SQL is not good for

- Consistent processing
  - Ignite SQL **is not transactional**
- AD-HOC SQL
  - Custom complex SQL query is likely to run long or even cause OOM

Ignite SQL is not good for

- Consistent processing
  - Ignite SQL **is not transactional**
- AD-HOC SQL
  - Custom complex SQL query is likely to run long or even cause OOM
- Query optimization
  - Ignite relies on H2, which is unaware of distributed specifics

Ignite SQL is not good for

- Consistent processing
  - Ignite SQL **is not transactional**
- AD-HOC SQL
  - Custom complex SQL query is likely to run long or even cause OOM
- Query optimization
  - Ignite relies on H2, which is unaware of distributed specifics
- Thoughtless usage of blocking operators
  - Ignite SQL accumulates intermediate query result in RAM
  - group by / order by over large table without condition can cause OOM
  - select over large table without condition can cause OOM

Agenda

- What is and what is not Ignite SQL: pros and cons
- Ignite SQL typical successful use cases
- How to cook Ignite SQL: four-step guide
- Ignite SQL: performance tuning
- Living with Ignite SQL: schema evolution

Separating side activities from main SQL processing

- Oracle / PostgreSQL / CRM is used for critical business processing
  - user transactions, money transfer

Separating side activities from main SQL processing

- Oracle / PostgreSQL / CRM is used for critical business processing
  - user transactions, money transfer
- Side business activities
  - end of day / month clearing
  - applications maintenance

Separating side activities from main SQL processing

- Oracle / PostgreSQL / CRM is used for critical business processing
  - user transactions, money transfer
- Side business activities
  - end of day / month clearing
  - applications maintenance
- Which are too resource intensive to run at main DB
  - may affect performance of critical operation

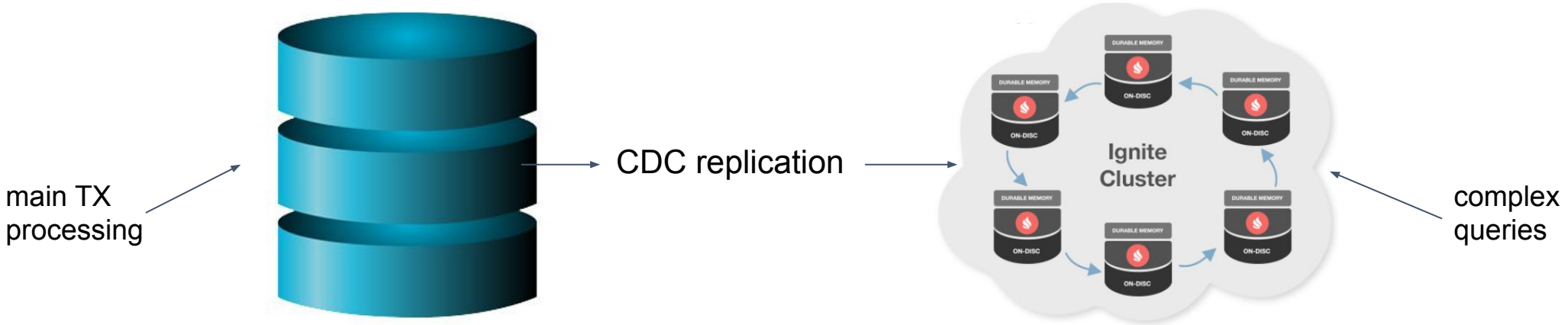
Separating side activities from main SQL processing

- Oracle / PostgreSQL / CRM is used for critical business processing
  - user transactions, money transfer
- Side business activities
  - end of day / month clearing
  - applications maintenance
- Which are too resource intensive to run at main DB
  - may affect performance of critical operation
- Or too expensive to run at main DB
  - may require buying additional instances
  - billing in some SaaS CRMs is bound to number of API calls

Separating side activities from main SQL processing

Solution:

- Keep your main business activities where they are
- Setup CDC between main database and Ignite SQL database
- Tune Ignite SQL to work well on your specific queries, enjoy the performance



Combining Ignite JCache and SQL

- Ignite JCache provides ACID distributed transactions
  - can be used for business critical processing

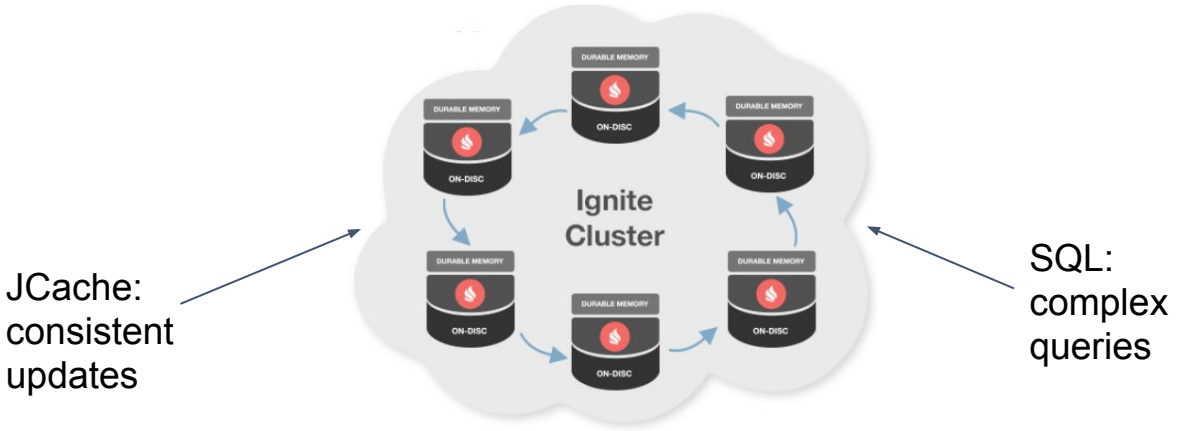
Combining Ignite JCache and SQL

- Ignite JCache provides ACID distributed transactions
  - can be used for business critical processing
  - not convenient for complex queries: joins, lookups, etc

Combining Ignite JCache and SQL

Solution:

- Access the same data both with JCache and SQL
  - JCache for consistent modification
  - SQL for complex queries where consistency under load is not critical



Agenda

- What is and what is not Ignite SQL: pros and cons
- Ignite SQL typical successful use cases
- How to cook Ignite SQL: four-step guide
- Ignite SQL: performance tuning
- Living with Ignite SQL: schema evolution

Step one: bootstrap your Ignite SQL schema

JCache-first way

- Mark your data classes with annotations

```
public class Person implements Serializable {
  /** Indexed field. Will be visible for SQL engine. */
  @QuerySqlField (index = true)
  private long id;

  /** Queryable field. Will be visible for SQL engine. */
  @QuerySqlField
  private String name;

  /** Will NOT be visible for SQL engine. */
  private int age;
```

Step one: bootstrap your Ignite SQL schema

JCache-first way

- Multi-fields and descending indexes are also supported

```
public class Person implements Serializable {
  /** Indexed in a group index with "salary". */
  @QuerySqlField(orderedGroups={@QuerySqlField.Group(
    name = "age_salary_idx", order = 0, descending = true)})
  private int age;

  /** Indexed separately and in a group index with "age". */
  @QuerySqlField(index = true, orderedGroups={@QuerySqlField.Group(
    name = "age_salary_idx", order = 3)})
  private double salary;
```

Step one: bootstrap your Ignite SQL schema

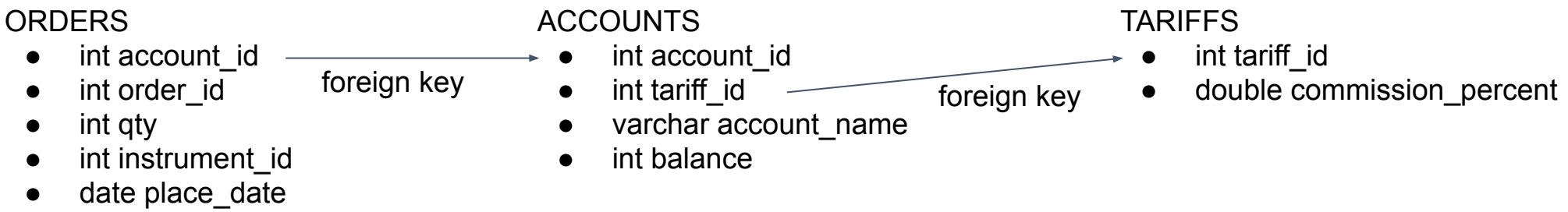
SQL-first way

- Ignite SQL supports creating table with DDL
- Template allows to specify JCache representation parameters

```
CREATE TABLE IF NOT EXISTS Person (
  age int, id int, city_id int, name varchar, company varchar,
  PRIMARY KEY (name, id))
  WITH "key_type=org.company.PersonId, value_type=org.company.PersonInfo"
```

Step two: prepare queries that should perform well in advance

Case: collect big quantity traders that use basic tariffs at the end of business day in order to prepare premium account offerings



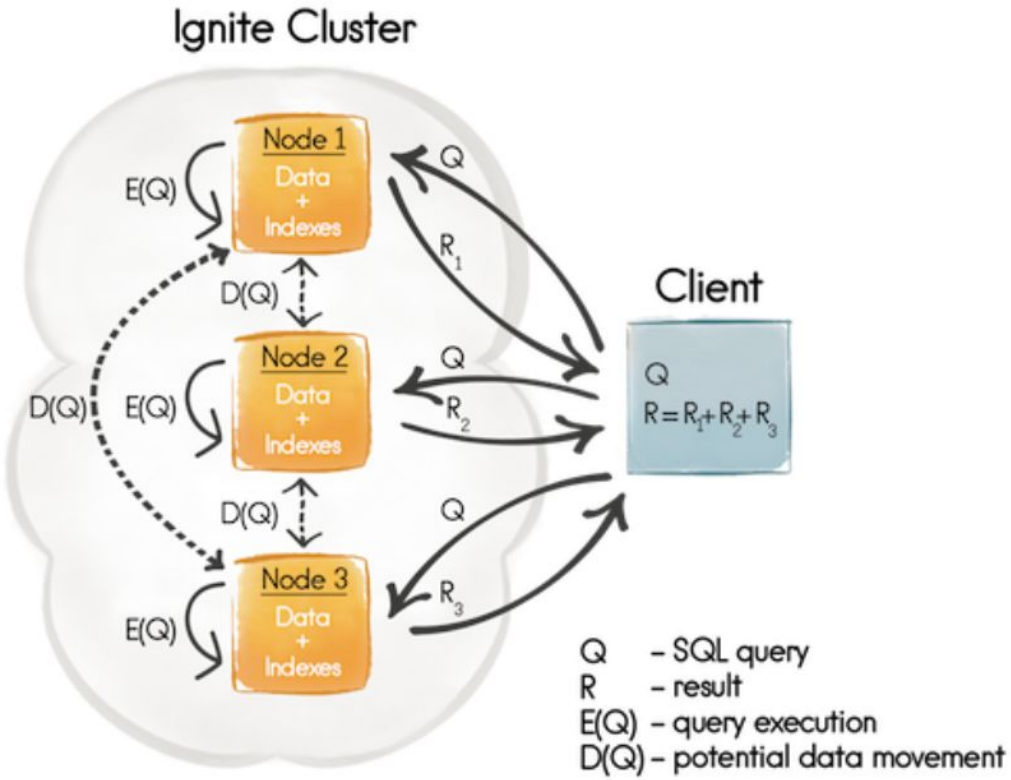

```
select account_name from ACCOUNTS A inner join ORDERS O on A.account_id = O.accound_id inner
join TARIFFS T on T.tariff_id = A.tariff_id where O.qty > 100 and T.comission_percent > 0.02
```

Step three: collocate your data

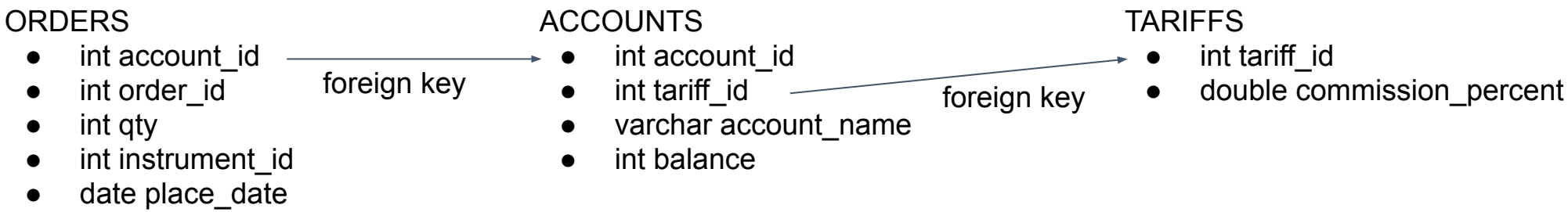
- By default, distributed joins work only when joined data is collocated

Step three: collocate your data

- By default, distributed joins work only when joined data is collocated
- You can specify `distributedJoins=true` parameter at your own risk
  - Join will work, but execution time and memory consumption may grow significantly



Step three: collocate your data



- ORDERS and ACCOUNTS table contain lots of data and joined on account_id
- Mark account_id with `@AffinityKeyMapped`
- TARIFFS has less data and can't be colocated along with ORDERS and ACCOUNTS
- Make cache for TARIFFS table REPLICATED

Step four: tune your performance

- Create indexes
  - Ignite supports only ordered B+ tree indexes
  - Indexing account_id and tariff_id would speed up join

Step four: tune your performance

- Create indexes
  - Ignite supports only ordered B+ tree indexes
  - Indexing account_id and tariff_id would speed up join
  - Use `inlineSize` to specify maximum number of index field bytes that will be inlined in B+ tree
    - If index is inlined, lookup won't require data row dereferencing and will be faster
    - `CREATE INDEX fast_city_idx ON sales (country, city) INLINE_SIZE 60;`

Step four: tune your performance

- Create indexes
  - Ignite supports only ordered B+ tree indexes
  - Indexing account_id and tariff_id would speed up join
  - Use `inlineSize` to specify maximum number of index field bytes that will be inlined in B+ tree
    - If index is inlined, lookup won't require data row dereferencing and will be faster
    - `CREATE INDEX fast_city_idx ON sales (country, city) INLINE_SIZE 60;`
- Proceed to fine-tuning
  - Ignite SQL provides various parameters to match your specific query

Agenda

- What is and what is not Ignite SQL: pros and cons
- Ignite SQL typical successful use cases
- How to cook Ignite SQL: four-step guide
- Ignite SQL: performance fine-tuning
- Living with Ignite SQL: schema evolution

Query parallelism

By default, every query is executed with one thread per node

- Can be changed with `cacheCfg.setQueryParallelism`
- Can't be changed in runtime
- Causes storage place overhead
  - every B+ tree is present in `queryParallelism` instances

Lazy flag

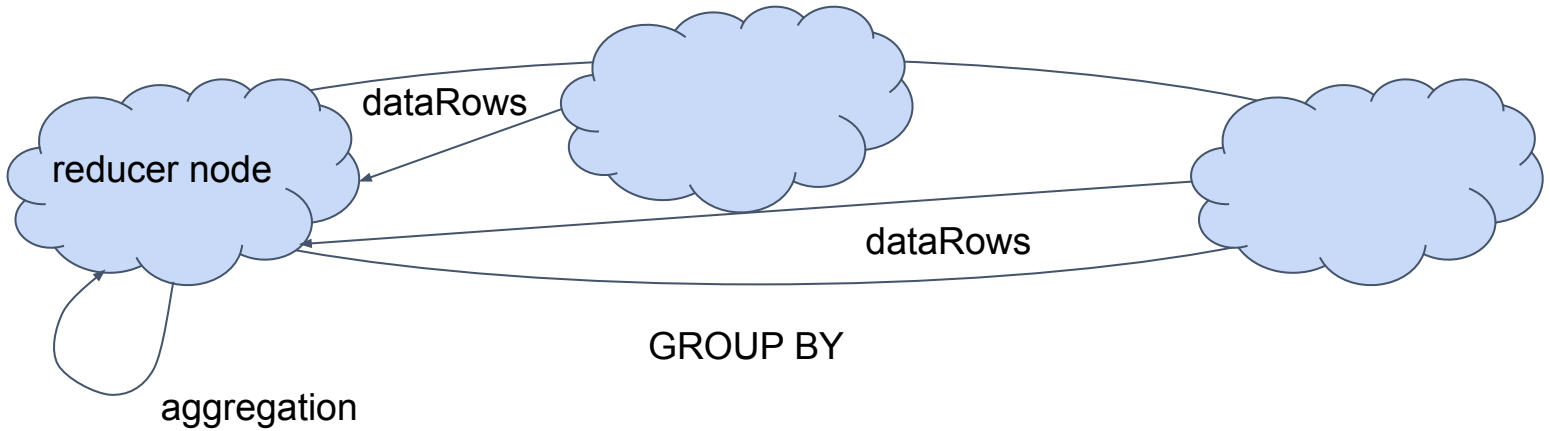
By default, the whole query result is composed in-memory on reducer

- Can be changed with `setLazy`
- Use lazy mode for cases when only part of the large result is needed
  - Query will be uploaded to reducer in batch on-demand mode
- Won't help for blocking operators
  - like order by / group by

Set collocated

By default, aggregation functions are considered as non-collocated

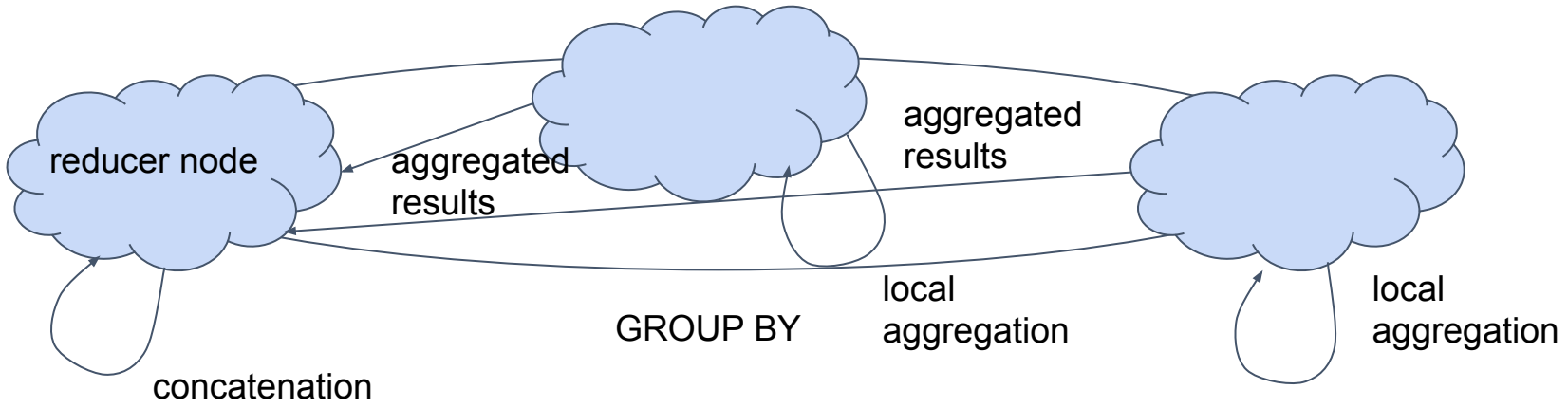
- Aggregation is performed on reducer



Set collocated

By default, aggregation functions are considered as non-collocated

- Aggregation is performed on reducer
- If you are sure that you perform aggregation on collocated field
  - Use `setCollocated`
- Collocation will be performed on mapped nodes, which is more scalable



Set IGNITE_SQL_MERGE_TABLE_MAX_SIZE if big aggregation result is expected

By default, aggregation functions are supported for up to 10000 various keys

- If you expect that aggregation on larger number of keys is possible
  - Override aforementioned system property
- Can’t be changed in runtime

Force tables join order

Don’t rely on Ignite query optimizer:

it doesn’t understand distributed specifics well

- Only nested loop joins are supported in Ignite
- Hash joins are present thought, but in experimental mode
- Use EXPLAIN PLAN on your queries
- If you are not satisfied with explained order
  - `setEnforceJoinOrder=true` will force joining in the order of mention

Agenda

- What is and what is not Ignite SQL: pros and cons
- Ignite SQL typical successful use cases
- How to cook Ignite SQL: four-step guide
- Ignite SQL: performance fine-tuning
- Living with Ignite SQL: schema evolution

Schema evolution

- You can add or remove columns from table

```
ALTER TABLE City ADD COLUMN IF NOT EXISTS population int;
ALTER TABLE Person DROP COLUMN (code, gdp);
```

- Storage data won't be changed
  - Only select (*) is affected
- Changing of column type is not supported
  - Remove column and add another with different name instead

Summary

- Ignite SQL will not serve you like
  “as good and universal as Oracle/Postgre, but distributed”

Summary

- Ignite SQL will not serve you like
  “as good and universal as Oracle/Postgre, but distributed”
- Planning your queries and configuring collocation in advance will bring decent performance, empowered by other Ignite features

Thanks for your attention!
Questions?

e-mail: irakov@gridgain.com

public list for discussions: user@ignite.apache.org

SQL documentation from GridGain:
https://www.gridgain.com/docs/latest/sql-reference/sql-reference-overview