Apache Ignite™

Tuning for Optimal Performance

Valentin Kulichenko
GridGain Lead Architect
Apache Ignite PMC

http://ignite.apache.org

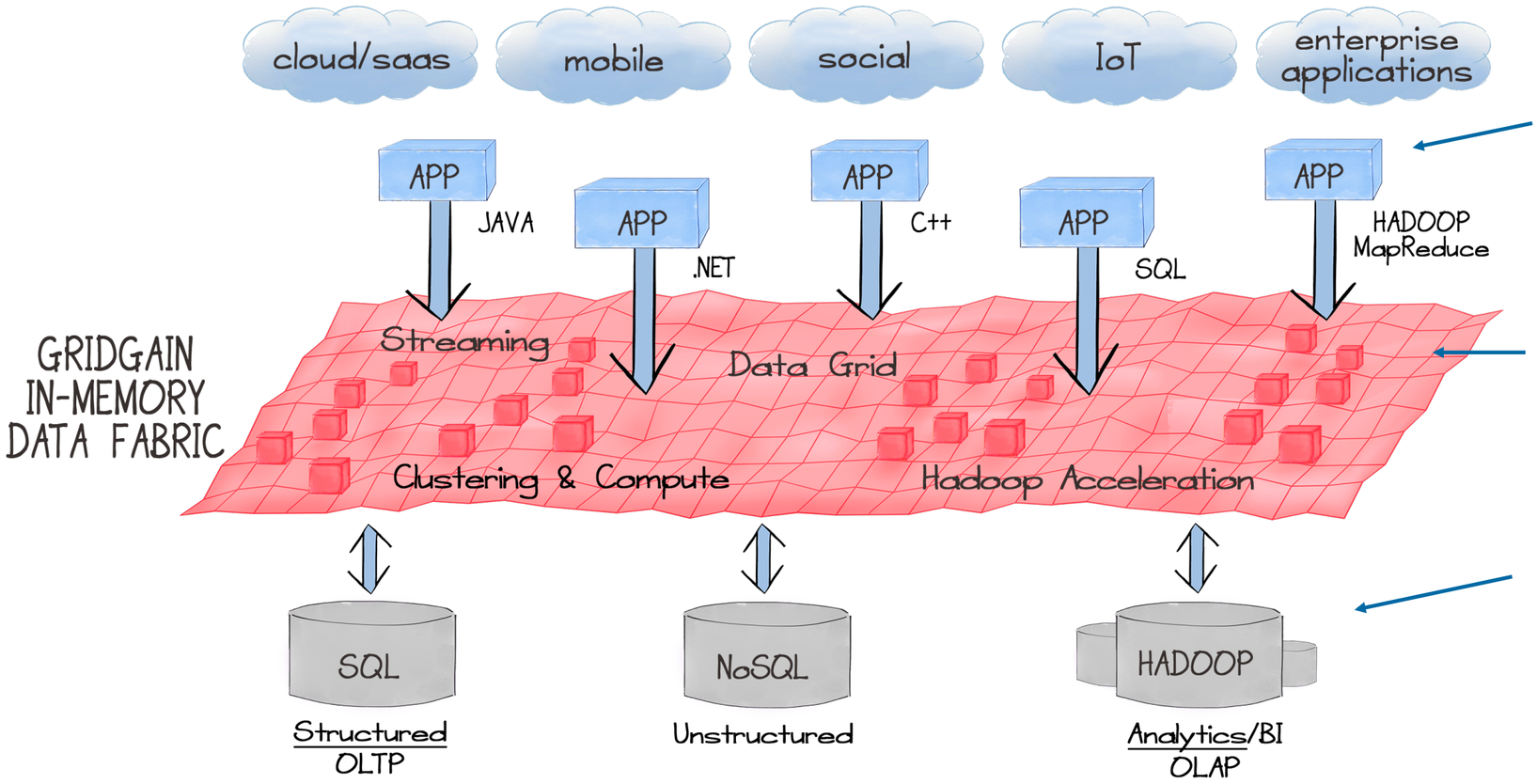
#apacheignite

GridGain

Agenda

- Apache Ignite Overview
- Performance Bottlenecks
- Basic Cache Operations
- Data Loading
- Affinity Collocation
- Tuning SQL Queries
- JVM Tuning
- Demo
- Q&A

Apache Ignite™ In-Memory Data Fabric: Strategic Approach to IMC



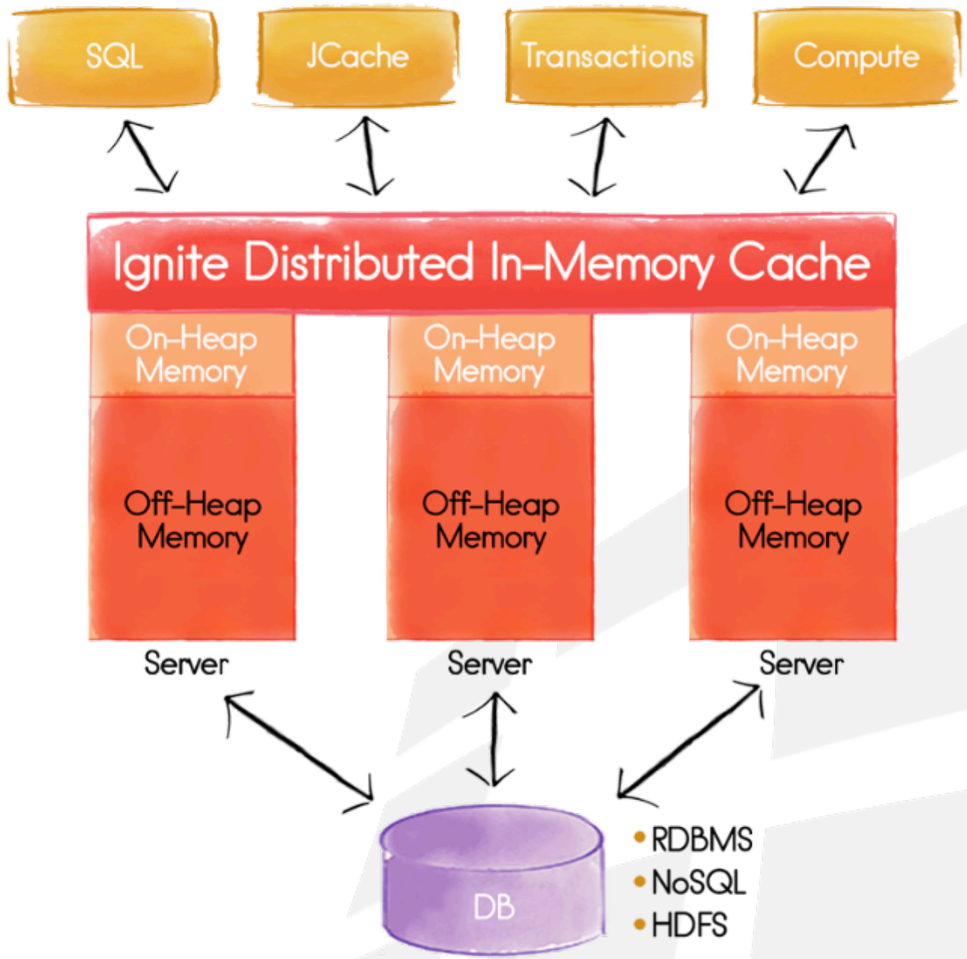
- Supports Applications of various types and languages

- Open Source – Apache 2.0
- Simple Java APIs
- 1 JAR Dependency
- High Performance & Scale
- Automatic Fault Tolerance
- Management/Monitoring
- Runs on Commodity Hardware

- Supports existing & new data sources
- No need to rip & replace

In-Memory Data Grid

- Distributed Key-Value Data Store
- Data Reliability
- High-Availability
  - Active replicas, automatic failover
- Data Consistency
  - ACID distributed transactions
- Distributed SQL
  - Advanced indexing



Performance Bottlenecks

- Application layer
  - Incorrect or ineffective code
- JVM layer
  - Insufficient memory
  - High CPU utilization by GC threads
  - Long GC pauses
- System layer
  - CPU saturation
  - Swapping
  - Disk or network I/O
- Hardware layer

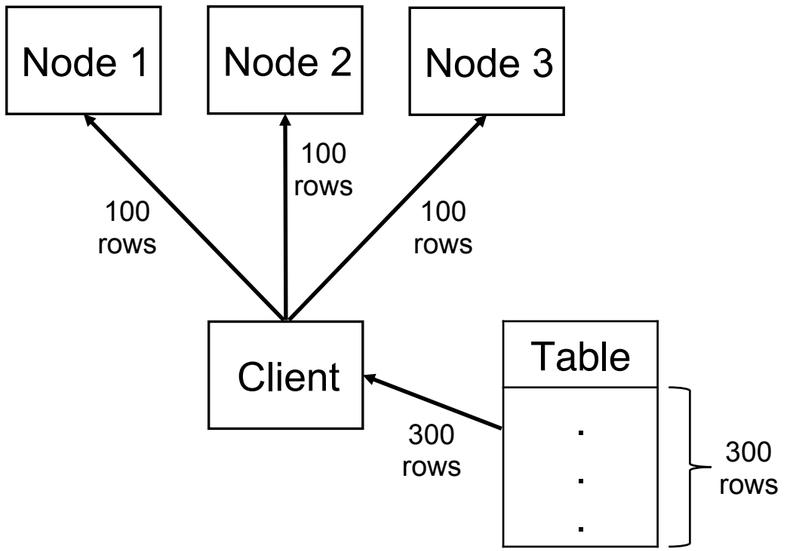
Basic Cache Operations

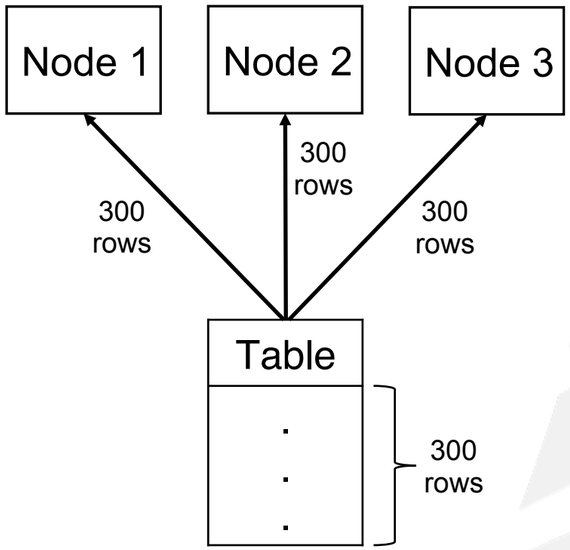
- get(), put(), remove(), …
- Latency vs throughput
- Try to batch to achieve higher throughput
- Manual batching: getAll(), putAll(), removeAll(), …
- Automatic batching: IgniteDataStreamer

```
try (IgniteDataStreamer<Integer, String> streamer = ignite.dataStreamer("my-cache")) {
    for (int i = 0; i < ENTRY_COUNT; i++)
        streamer.addData(i, Integer.toString(i));
}
```

Data Loading

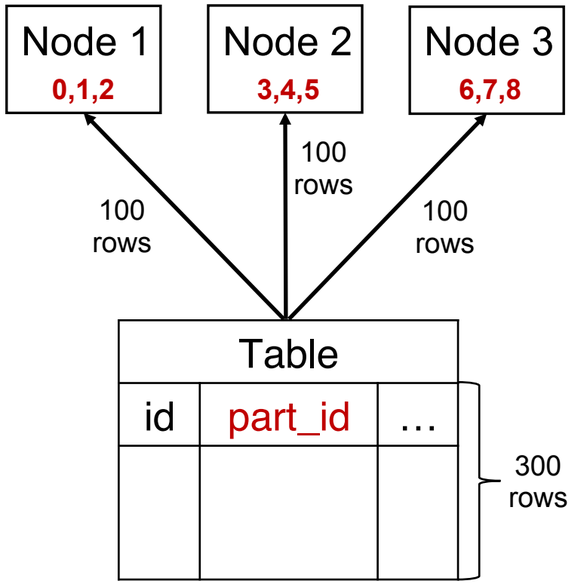
- IgniteDataStreamer
  - Pros: single node connecting to the database
  - Cons: data has to be transferred through network to target nodes
- CacheStore.loadCache(..)
  - Pros: no network communication between nodes
  - Cons: each node has to load the whole dataset





Partition Aware Data Loading

- Use CacheStore.loadCache(..)
- Reduce network communication with the database
- Performs well, but requires database schema change



Node 1: select from Table where part_id in (0,1,2)
Node 2: select from Table where part_id in (3,4,5)
Node 3: select from Table where part_id in (6,7,8)

Affinity Collocation

- Affinity function and affinity key
- Cache key ➡ Affinity key ➡ Partition number ➡ Node
- Collocate as much as possible
  - Data with data
  - Computations with data

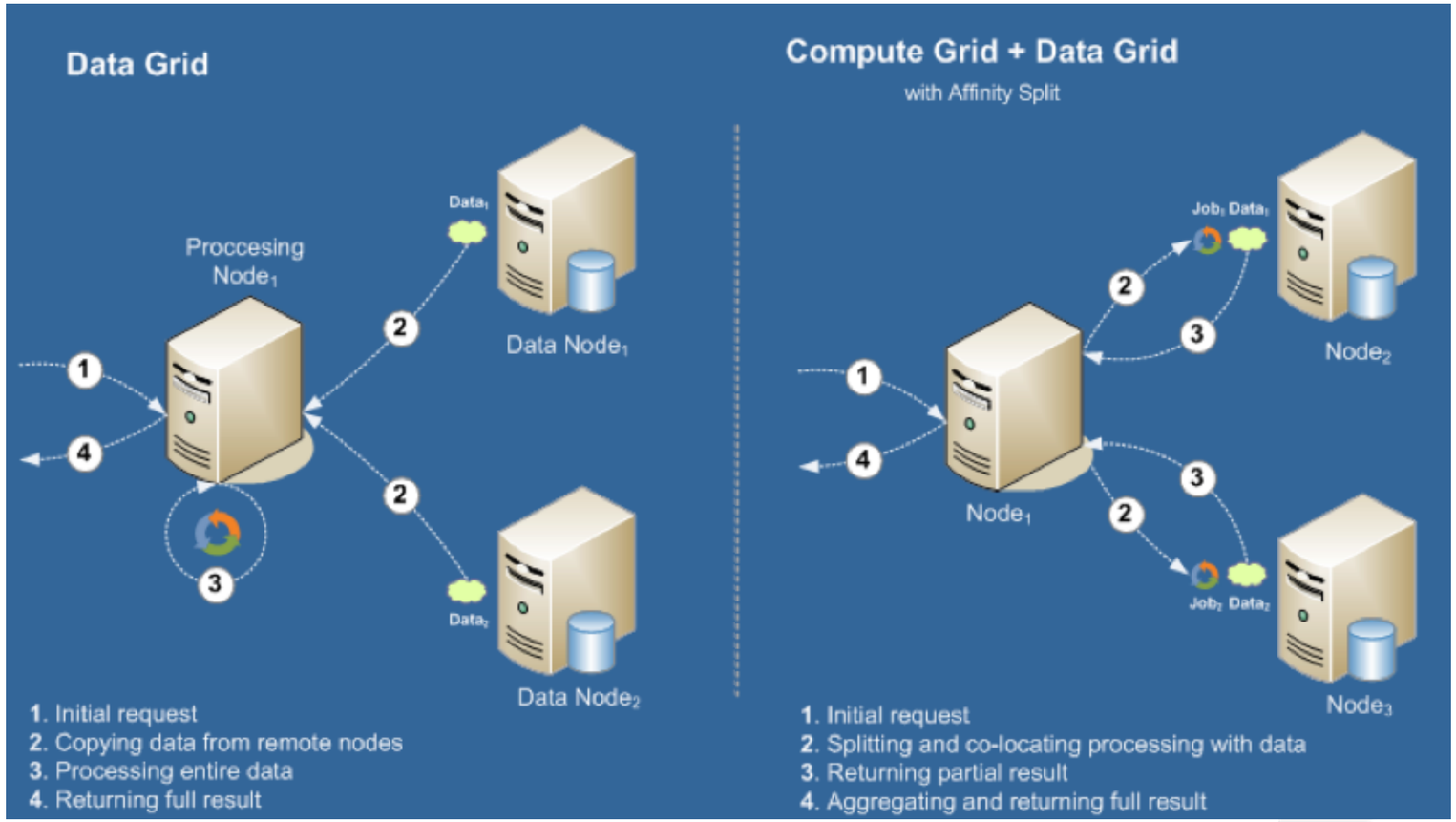
```
public class EmployeeKey {
    private int id;

    @AffinityKeyMapped
    private int organizationId;
}
```

- **id** is the cache key (unique for all employees)
- **organizationId** is the affinity key (can be the same for different employees)

Client-Server vs Affinity Collocation

Client-Server



Affinity Colocation

Tuning SQL Queries

- General Optimizations
  - Don’t index everything
  - Use group indexing
  - Adjust page size
- Off-Heap Optimizations
  - Tune on-heap row cache size
- Run EXPLAIN statement
- GridGain WebConsole
  - Query Execution & Debugging
  - Query Monitoring
- H2 Debug Console
  - Low level debugging

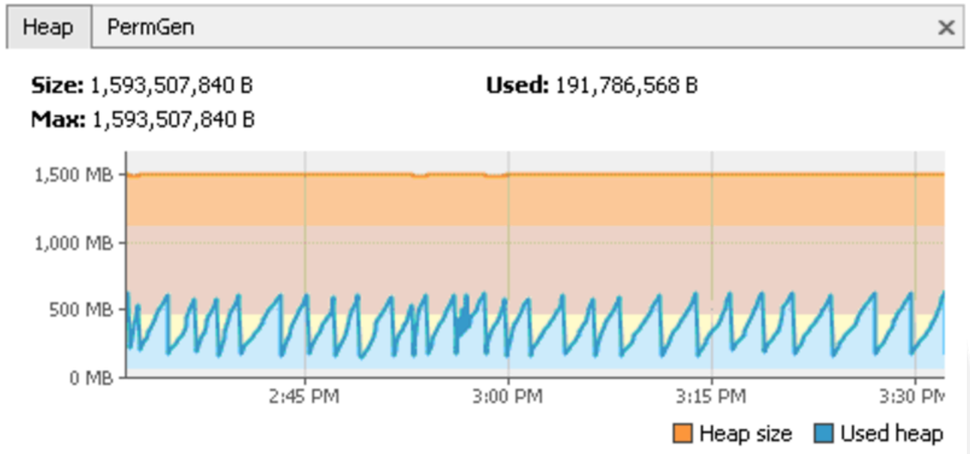
https://apacheignite.readme.io/docs/sql-queries

Collocated vs non-collocated SQL joins

- Collocated mode
  - Pros: no data movement between nodes, better performance
  - Cons: data has to be collocated in advance
- Non-collocated mode
  - Pros: no need to collocate data
  - Cons: potential data movement between nodes
- Use cases
  - Use collocated mode as much as possible
  - Use non-collocated mode as the last resort

JVM Tuning

- Avoid large heap sizes, use off-heap memory
- Use G1 or CMS collectors
- Always collect GC logs
- In case of OOME – analyze heap dump
- Use VisualVM and FlightRecorder



https://apacheignite.readme.io/docs/jvm-and-system-tuning

Demo

ANY QUESTIONS?

Thank you for joining us. Follow the conversation.

http://ignite.apache.org

 #apacheignite

GridGain