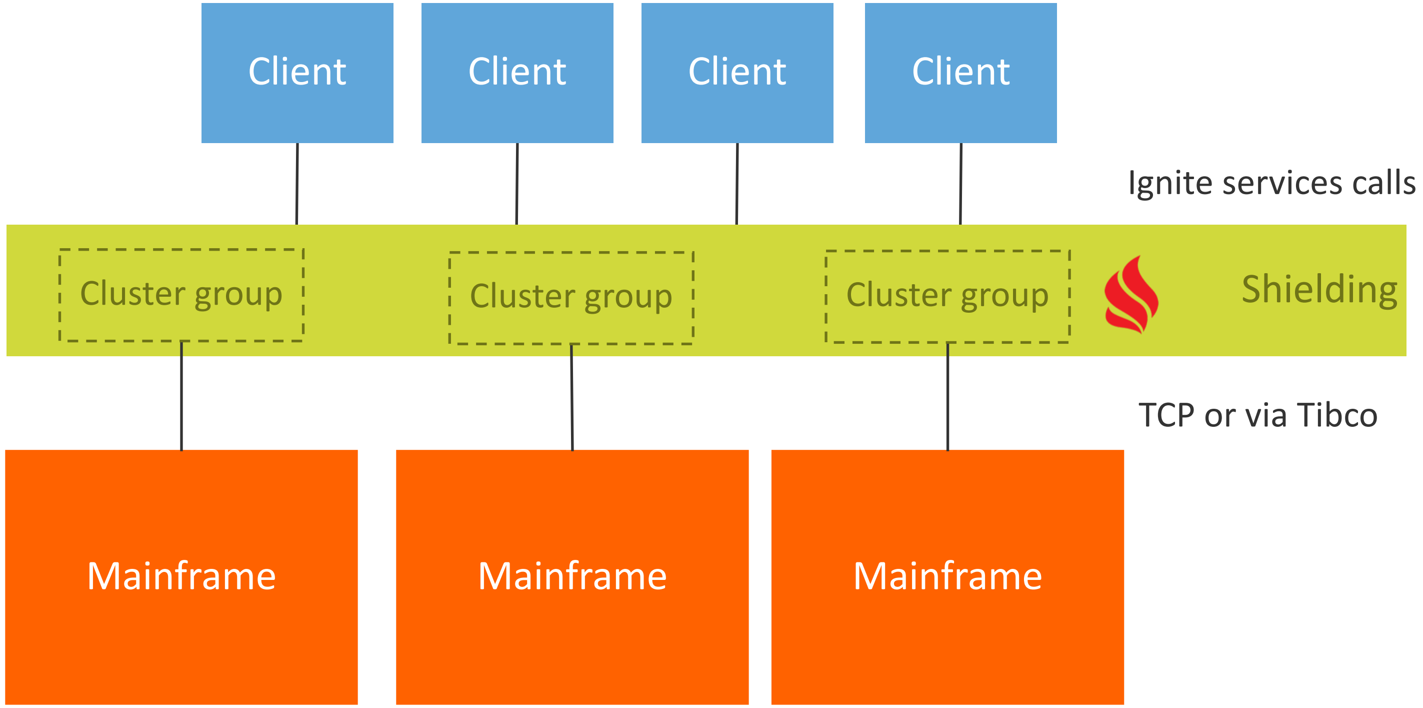
In-Memory Computing SUMMIT | NORTH AMERICA 2018

Exploring the Potential of Ignite Using Classless Design

David Follen
ING

Original use case: ShieldING

- Layer in front of the mainframes
- Serves many applications
- Caches data to shield mainframe from parallel concurrent load
- Big cluster own by different teams: multitenant



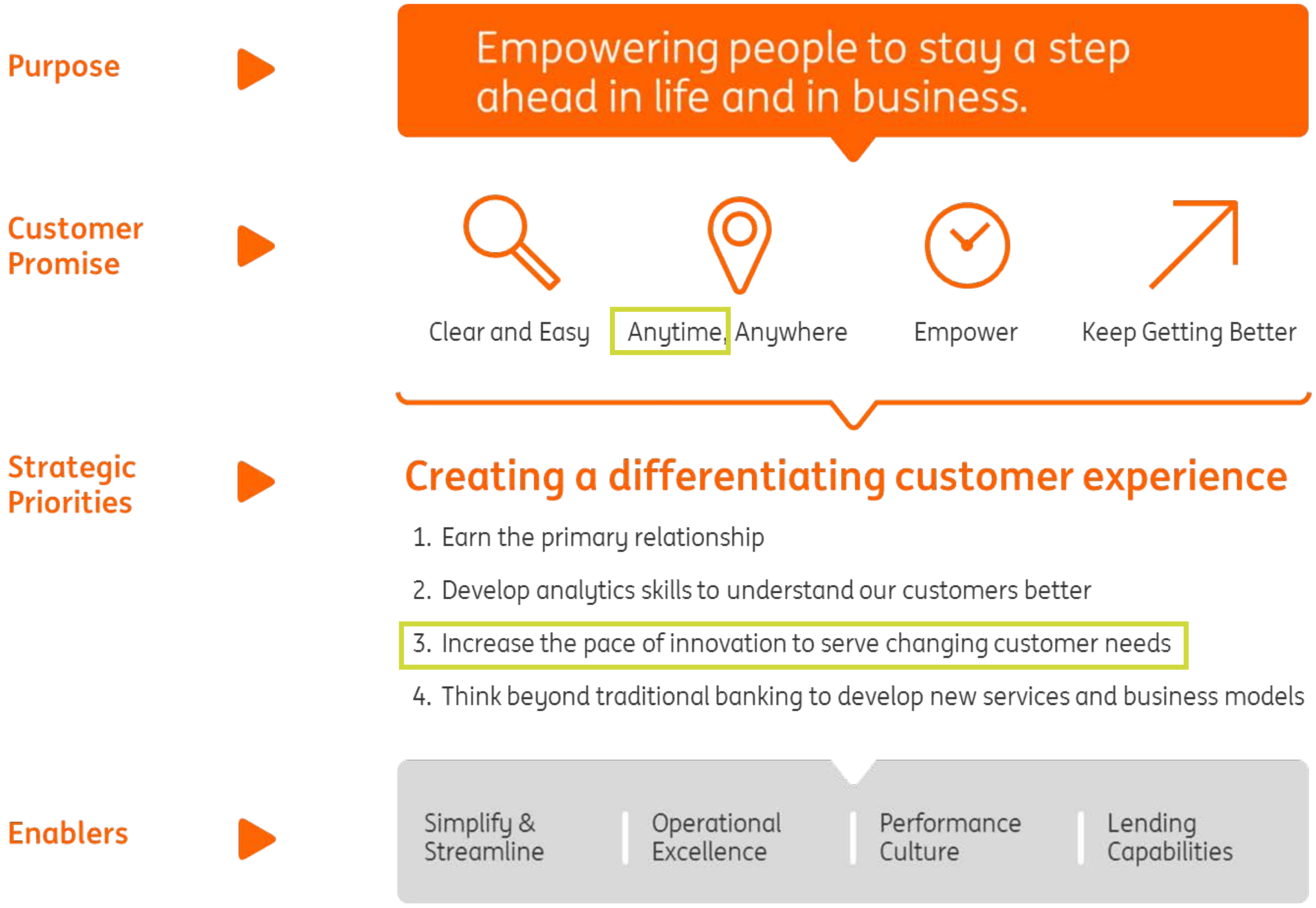
Investing in Ignite

- In-Memory Computing shows promising features
  - Resilient
  - Performant
  - Scalable
  - High availability
  - Consistency

- Ignite showed some limitations
  - Service grid present many issues
    - Update of a service imposes a full restart of the grid
    - Issues with services lifecycle

  - Multi-tenancy complexities
    - Configuration is propagated on all nodes
    - missing/incompatible classes might result on impossibility to start the node

ING’s Think Forward strategy



Purpose

Empowering people to stay a step ahead in life and in business.

Customer Promise

Clear and Easy | Anytime, Anywhere | Empower | Keep Getting Better

Strategic Priorities

Creating a differentiating customer experience

1. Earn the primary relationship
2. Develop analytics skills to understand our customers better
3. Increase the pace of innovation to serve changing customer needs
4. Think beyond traditional banking to develop new services and business models

Enablers

Simplify & Streamline | Operational Excellence | Performance Culture | Lending Capabilities

Scenario

Define requirements for an application
Come up with a design
Introduce changes in the requirements

Payment application

- User can get list of accounts and their balances

- User can get a list of transactions of an account

- User can initiate a debit from an account
  - Debit currency has to be identical to account currency
  - Debit amount has to be lower or equal to account balance

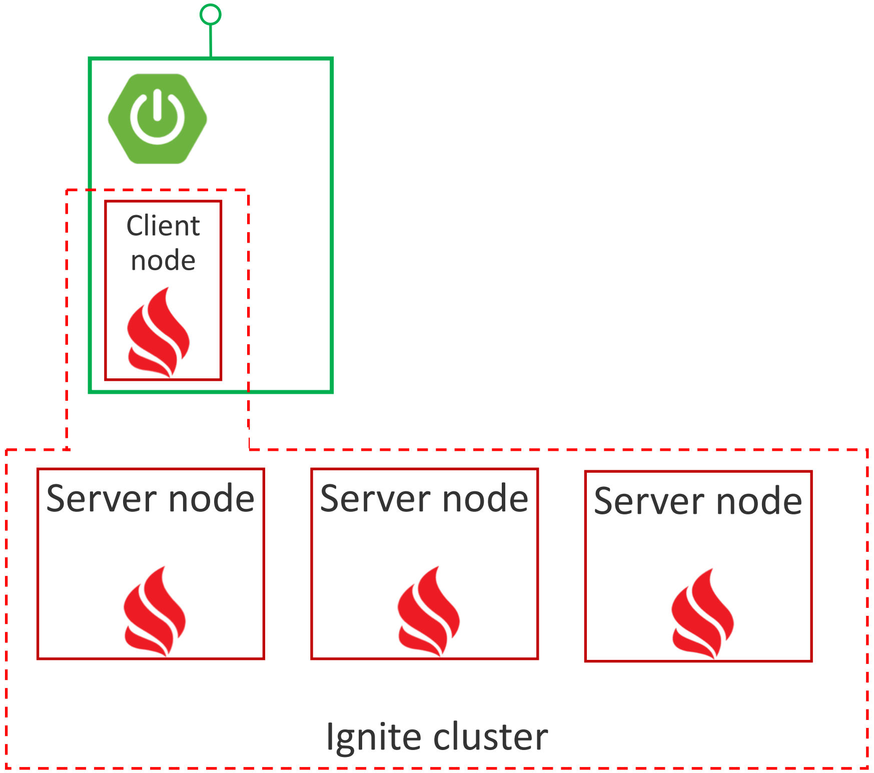
- Focus on backend
- Expose REST API
- Simple application (no authorisation/authentication)

Architecture

- Vanilla Ignite server nodes
- Ignite Native persistent store

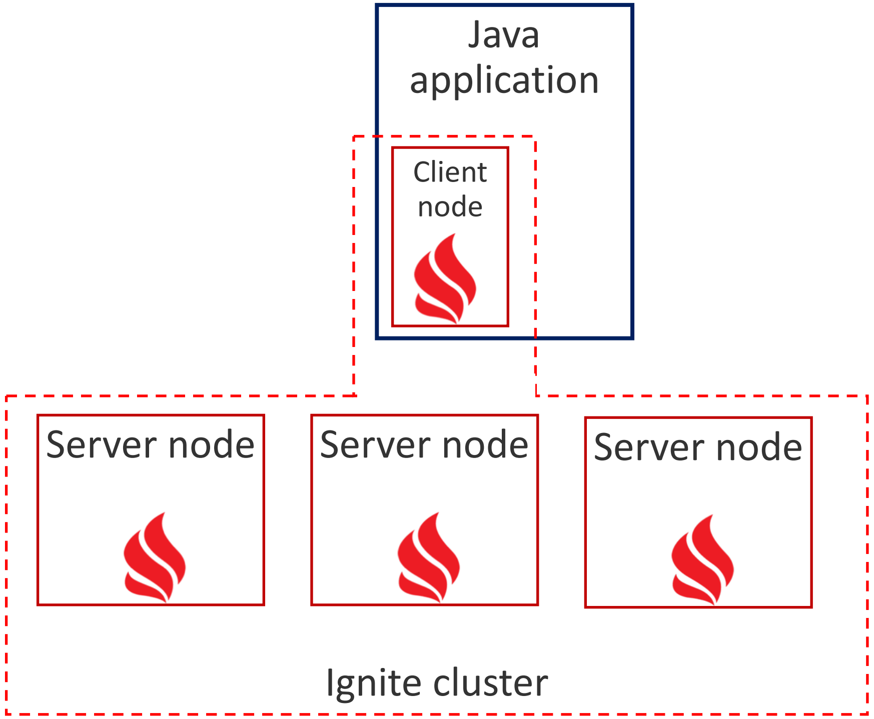
- Springboot based REST server
- Springboot server starts an Ignite client node

- Stable server node topology
- Allows scaling of the API layer independently of the data/compute layer
- Ignite cluster can be seen as datastore



Creating Ignite caches

- Maintenance of the data store
- Done via simple Java application connecting via an Ignite Client node
  - Create caches
  - Create/update indexes



Model

Customer cache

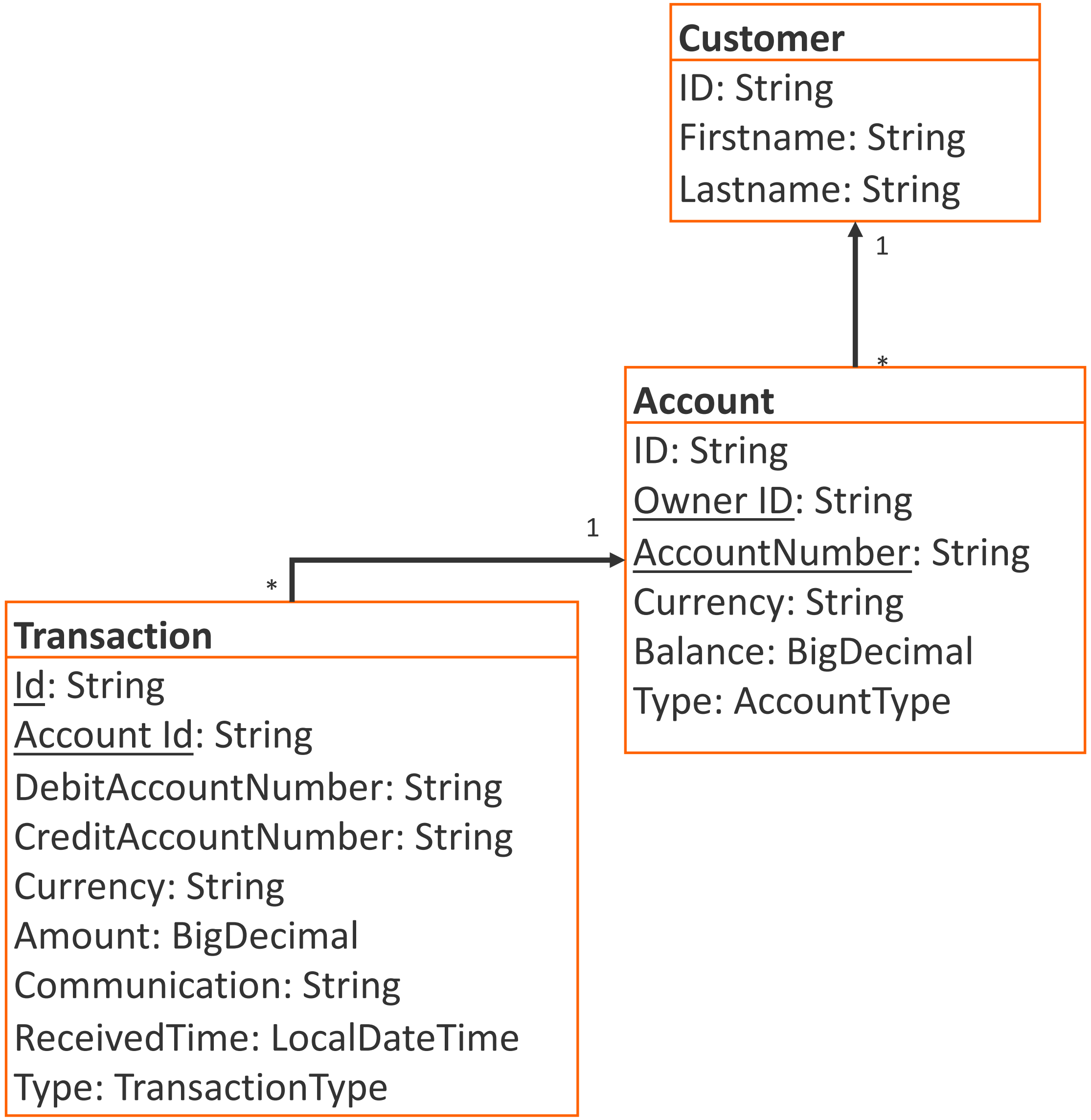
- PARTITIONED
- Backup: 1

Account cache

- PARTITIONED
- Backup: 1
- Transactional
- Index on Owner Id
- Index on AccountNumber

Transaction cache

- PARTITIONED
- Backup: 1
- Transactional
- Index on AccountId



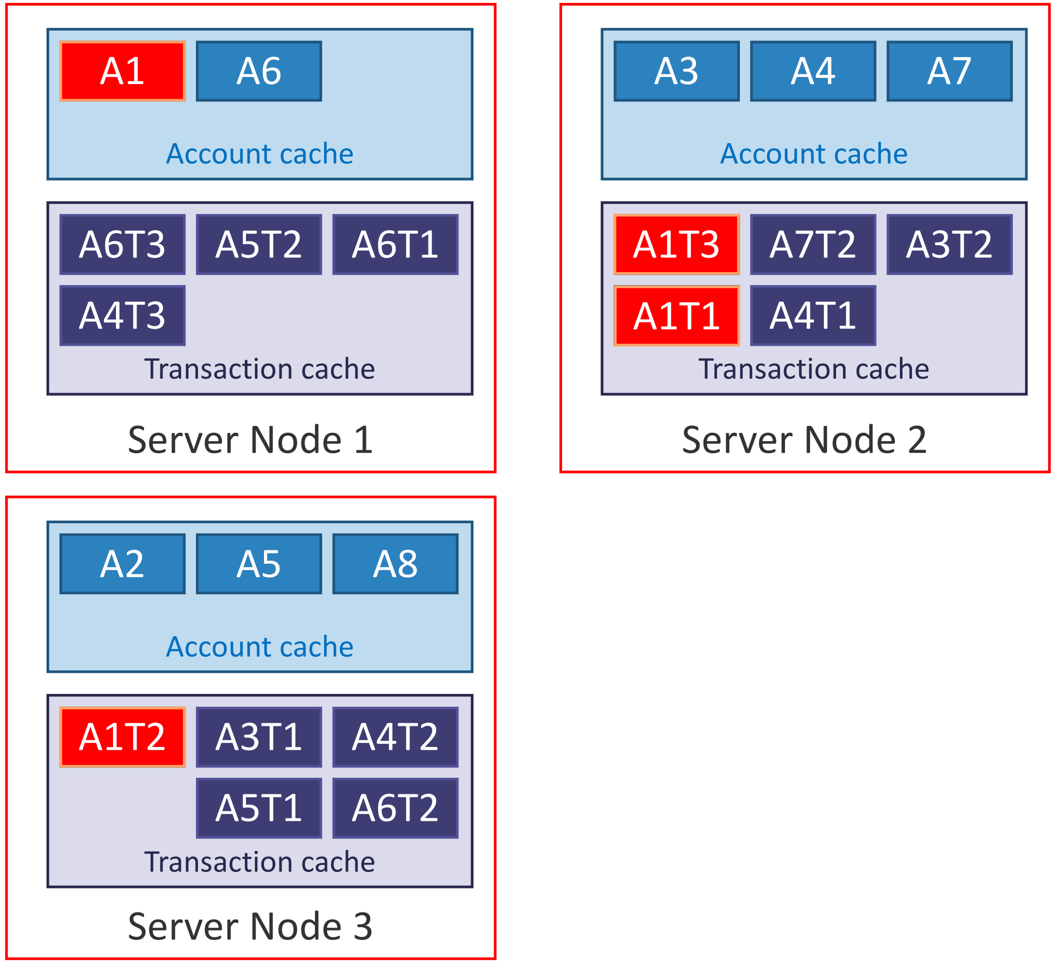
Customer
ID: String
Firstname: String
Lastname: String

Account
ID: String
Owner ID: String
AccountNumber: String
Currency: String
Balance: BigDecimal
Type: AccountType

Transaction
Id: String
Account Id: String
DebitAccountNumber: String
CreditAccountNumber: String
Currency: String
Amount: BigDecimal
Communication: String
ReceivedTime: LocalDateTime
Type: TransactionType

Data affinity co-location

No co-location



Server Node 1

Account cache: A1, A6

Transaction cache: A6T3, A5T2, A6T1, A4T3

Server Node 2

Account cache: A3, A4, A7

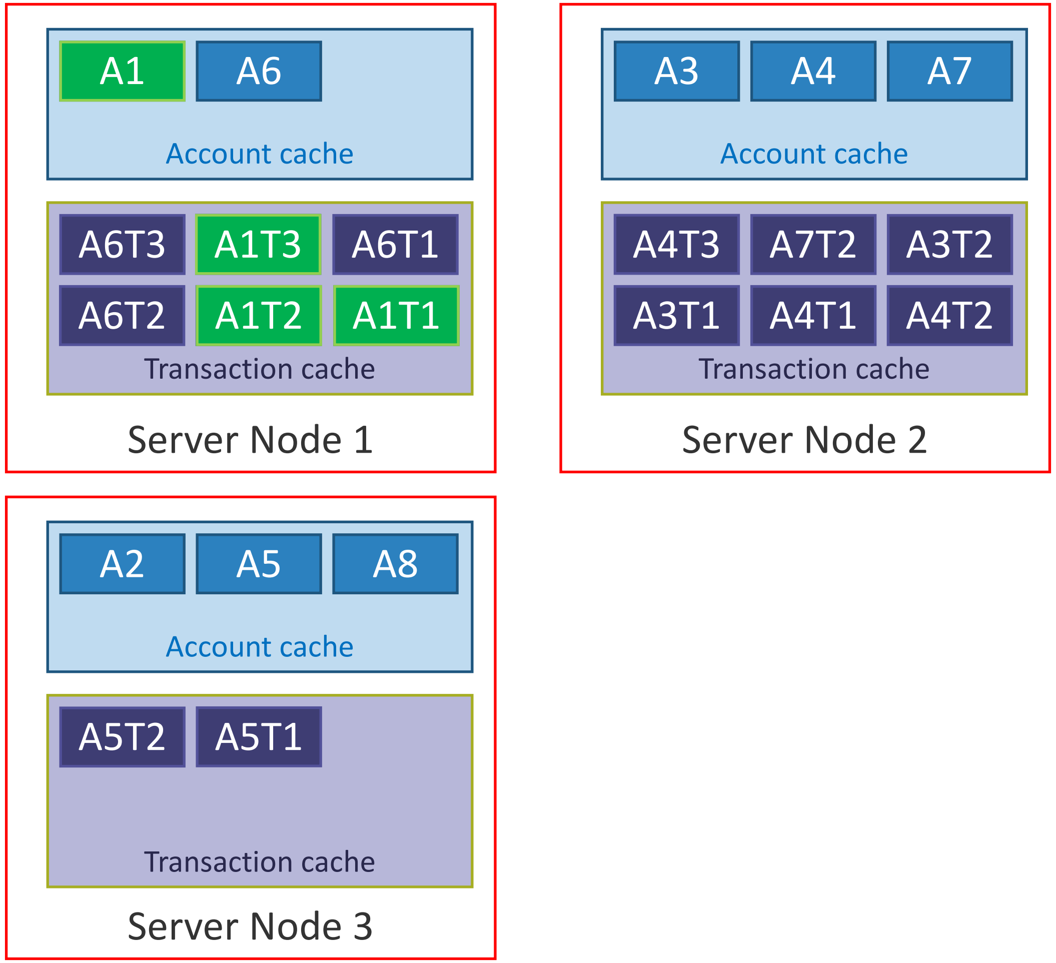
Transaction cache: A1T3, A7T2, A3T2, A1T1, A4T1

Server Node 3

Account cache: A2, A5, A8

Transaction cache: A1T2, A3T1, A4T2, A5T1, A6T2

Co-location



Server Node 1

Account cache: A1, A6

Transaction cache: A6T3, A1T3, A6T1, A6T2, A1T2, A1T1

Server Node 2

Account cache: A3, A4, A7

Transaction cache: A4T3, A7T2, A3T2, A3T1, A4T1, A4T2

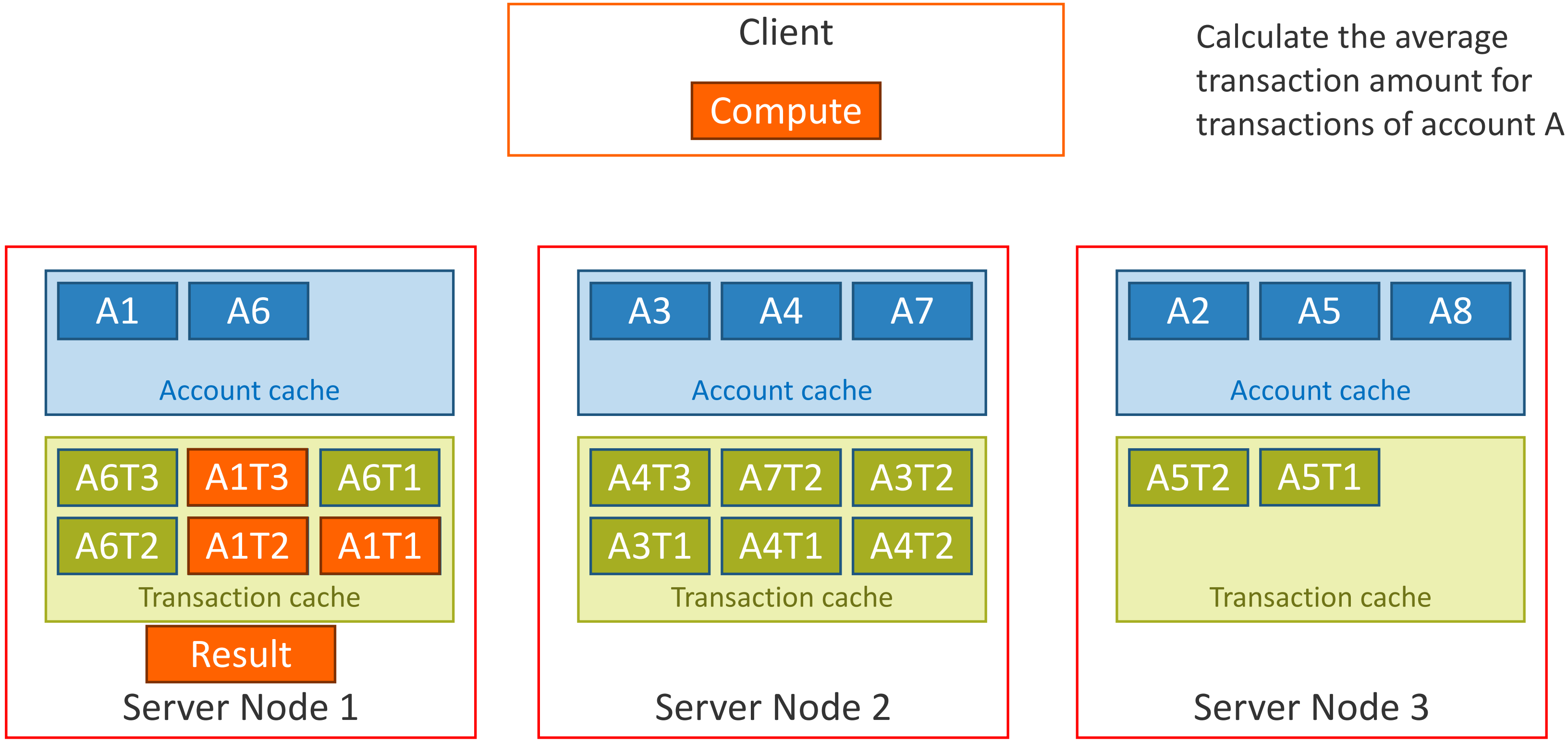
Server Node 3

Account cache: A2, A5, A8

Transaction cache: A5T2, A5T1

Affinity execution

Execute the code along with the data

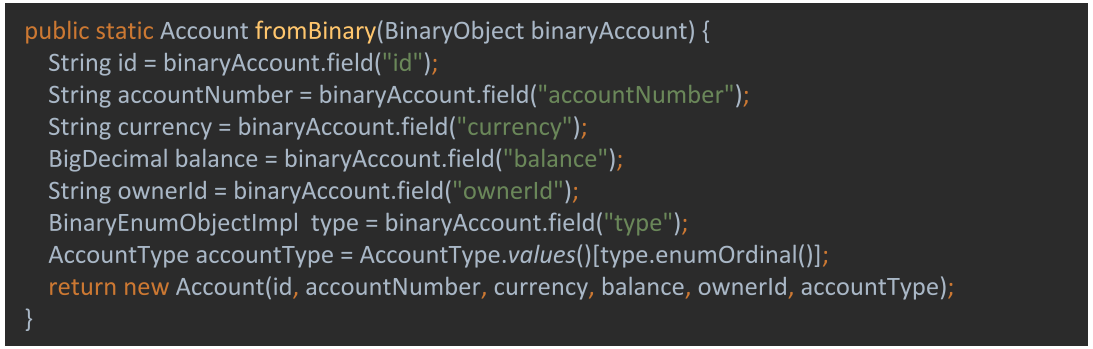


Calculate the average transaction amount for transactions of account A1

See talk from Valentin Kulichenko (Gridgain) from IMC Summit EU 2018:
https://www.imcsummit.org/2018/eu/session/want-extreme-performance-scale-do-distributed-right-way

Working with BinaryObject

- Do not deploy business nor model classes on Ignite server nodes
- Any client can connect, no classpath/version/dependency conflict
- Only works with BinaryObjects (see https://apacheignite.readme.io/docs/binary-marshaller)
- Puts de-serialisation on the client application



```
public static Account fromBinary(BinaryObject binaryAccount) {
   String id = binaryAccount.field("id");
   String accountNumber = binaryAccount.field("accountNumber");
   String currency = binaryAccount.field("currency");
   BigDecimal balance = binaryAccount.field("balance");
   String ownerId = binaryAccount.field("ownerId");
   BinaryEnumObjectImpl  type = binaryAccount.field("type");
   AccountType accountType = AccountType.values()[type.enumOrdinal()];
   return new Account(id, accountNumber, currency, balance, ownerId, accountType);
}
```

Demo code (a)

- Ignite cluster
  - No dependency except ignite jars
  - Starts nodes

- API server
  - Springboot based application
  - Exposes REST endpoints
  - Uses a client node to connect to the cluster

- Maintenance client
  - Simple java application
  - Uses a client node to connect to the cluster

Let’s accept new requirements

- Users can choose to receive an alert when account balance goes under a given amount
  - Limit amount must be > 0
- When a debit is received, if the resulting amount is below the alert amount, an alert is sent to the customer

Create a new cache for outgoing alerts

Add a new field on the Customer: contact details

Add a new field on the Account: alertAmount

Application evolution

- Keep ignite cluster up and running
  - No restart of server nodes: no rebalancing management

- Use the migration client to create the new Alert cache
- Validation of the transaction is done on Ignite server with a compute task
- Start a different API server
- In the service to update the limit amount, we also ask for the contact details
  - Customers who use this service will be represented by a different model
  - Existing applications will be able to continue reading the data
  - New applications need to deal with customers that are migrated yet

Application evolution

Alert cache

- PARTITIONED
- Backup: 1

New field on Customer:
ContactDetails: String

New field on Account:
Limit: BigDecimal

Alert

ID: String
Destination: String
Message: String
CreationTime: LocalDateTime

Demo code (b)

Keep existing running

- Second API server
  - Copy of the first one with modifications for the new requirements

What we achieved

- No need to restart the cluster to update the application
- Have multiple clients with different concerns
- Used co-location for best performance

Using binary objects and class-less design we managed to solve the issues we had encountered

Solution limitations

- Only application owner of the data should modify the data
- Mainly works with Ignite native persistence
- More effort to work with BinaryObjects
- Does not work with Ignite Queues or Topics
- Once a cache is created, query fields are fixed (Schema-on-write vs Schema-on-read)

ing.be/jobs

https://www.linkedin.com/company/ing/
https://www.linkedin.com/in/david-follen/

✉ david.follen@ing.com

ING