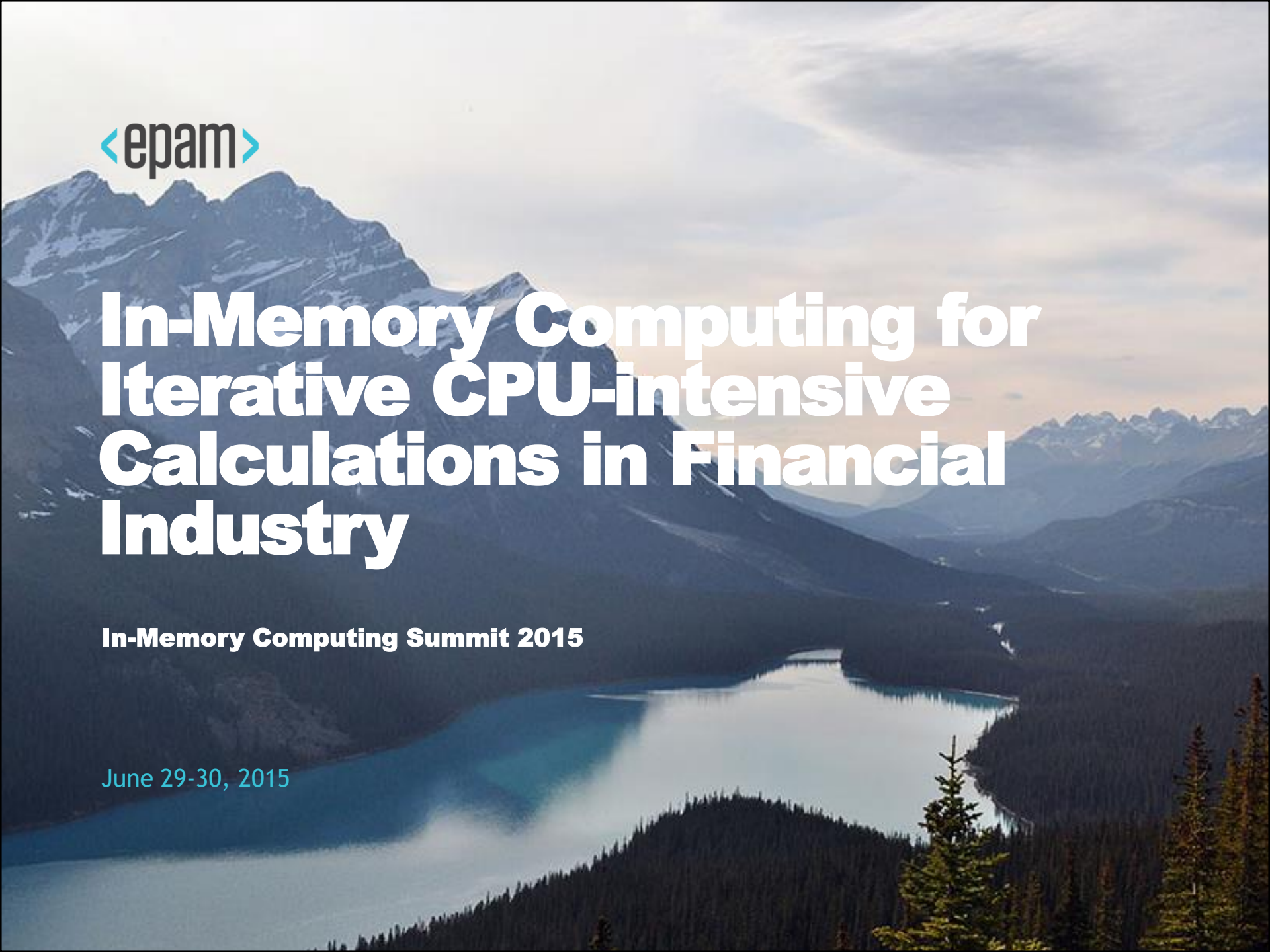


In-Memory Computing for Iterative CPU-intensive Calculations in Financial Industry

In-Memory Computing Summit 2015

June 29-30, 2015

Contacts

Alexandre Boudnik

Senior Solution Architect, EPAM Systems

Alexandre_Boudnik@epam.com

EPAM Big Data Competency Center

OrgCompetencyBigData@epam.com

Who I am

- I used to work on software development tools like compilers, hardware emulators, and testing tools for real-time systems
- My first American job was to lead development of SQL query engine for MPP grid in early 2000
- Now I'm working with many of the Big Data and Fast Data aspects:
  - Hadoop ecosystem for enterprise (data governance, lineage)
  - Streaming, complex event processing, in-memory caches and real-time graphs processing
  - Building the scalable, fault tolerant distributed systems on clusters

The problem our customers have faced

- Out clients in financial advisers want:
  - To get support for complex iterative calculations
  - To get a faster response than conventional tools provide
  - To have better (faster) support for what-if scenarios
  - To call calculations from web-based tools and from MS Excel
- Drawbacks of conventional BI tools and SQL technologies:
  - Not enough expandability (it is hard to use custom code)
  - Scalability, Concurrency and Response time

Techniques and Practices to Develop

Using in-memory technologies for financial calculations:

- Generalized approach to parallel recursive calculations on a grid
- Memory-efficient object data model to reduce memory footprint
- OData (hierarchal data model) on top of object data model

Wealth Management

Use case: Wealth Management Organizations

To manage client portfolios effectively, organizations need to calculate various financial functions:

- Over wide period of time
- For many financial vehicles (instruments)
- Using multiple hierarchal user-defined groupings (roll-up)
- Providing ability to compare performance with standard or user defined benchmarks
- Providing comfortable user experience (response time)
- Near-real time what-if scenarios
- Providing concurrent access for hundreds of users

Traditional SQL-based tools: road blocks

- SQL-based BI tools out-of-the box:
  - Are ineffective in supporting complex calculation defined on hierarchal structures along time dimension
  - Don’t provide sufficient support for complex calculations
  - Provide limited scalability
- Using the stored procedures to expand functionality is not an perfect answer:
  - Can be used for iterative calculations
  - High-Level languages like java and C# are available for some platforms
  - The differences in stored procedure APIs lead to vendor lock

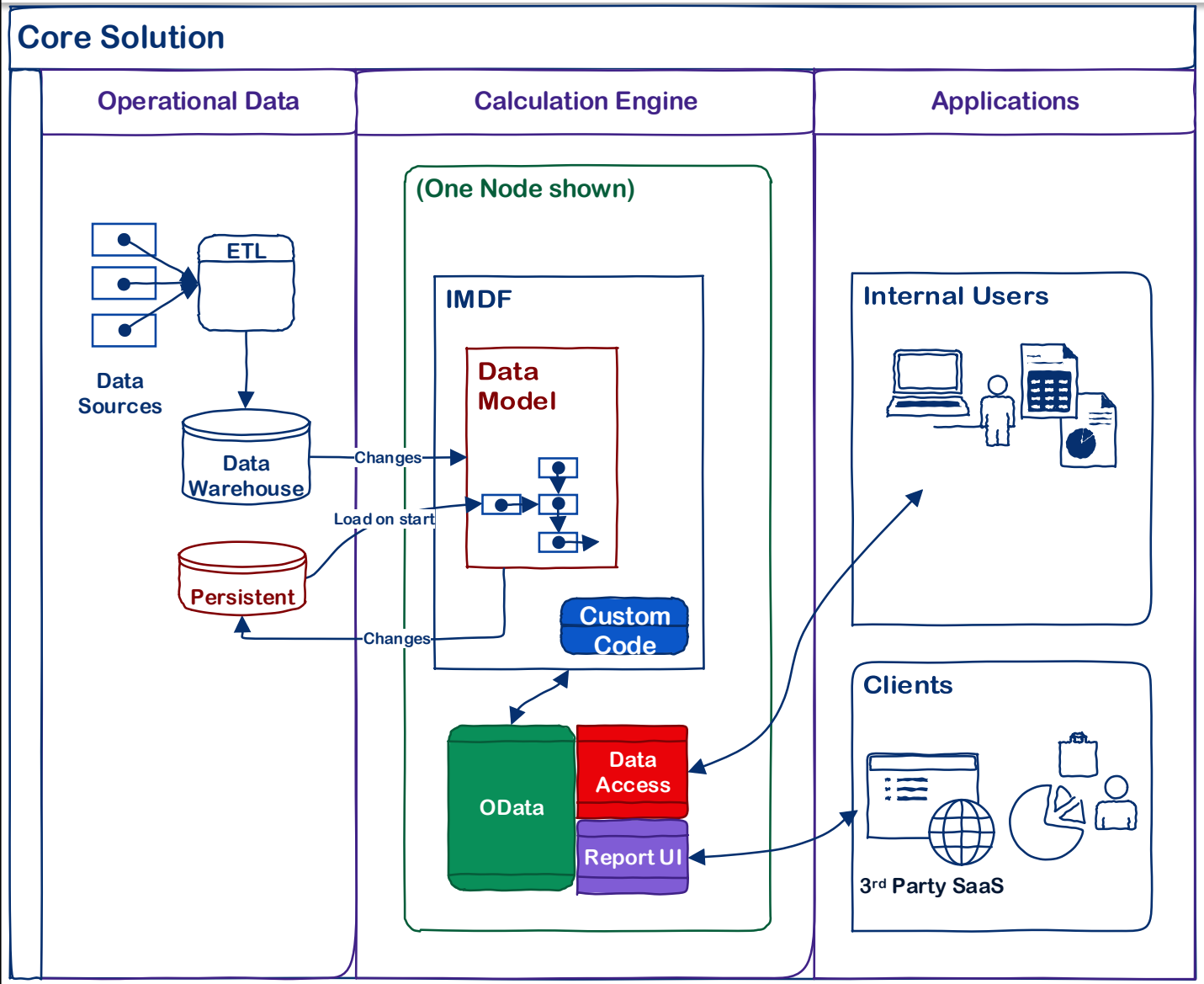
In-Memory Data Grid: getting out of the woods

- Architecture based on Distributed In-Memory Data Grid provides high performance, quick response and scalability in terms of concurrent load, amount of data, and complex calculations
- There are many implementations, both proprietary and open-source, with same computation models and very similar API
- Vendors we had considered:
  - GridGain, Gigaspaces XAP, SAP Hana, MS SQL Server In-Memory 2012 and 2014, Aerospike, Clustrix, Tibco ActiveSpaces, ScaleOut Software, Pivotal's gemfire/sqlfire, Oracle Coherence, MongoDB, Haselcast

Rationales for In-Memory Data Grid

- Speed-up the complex iterative CPU-intensive financial calculations:
  - Fast access to in-memory data
  - Fast execution of individual requests by implementing parallel versions of calculation algorithms
- Achieve fast response under highly concurrent load:
  - Load balancing
  - Parallel query execution
  - Distributed calculations

Core Solution: Architecture



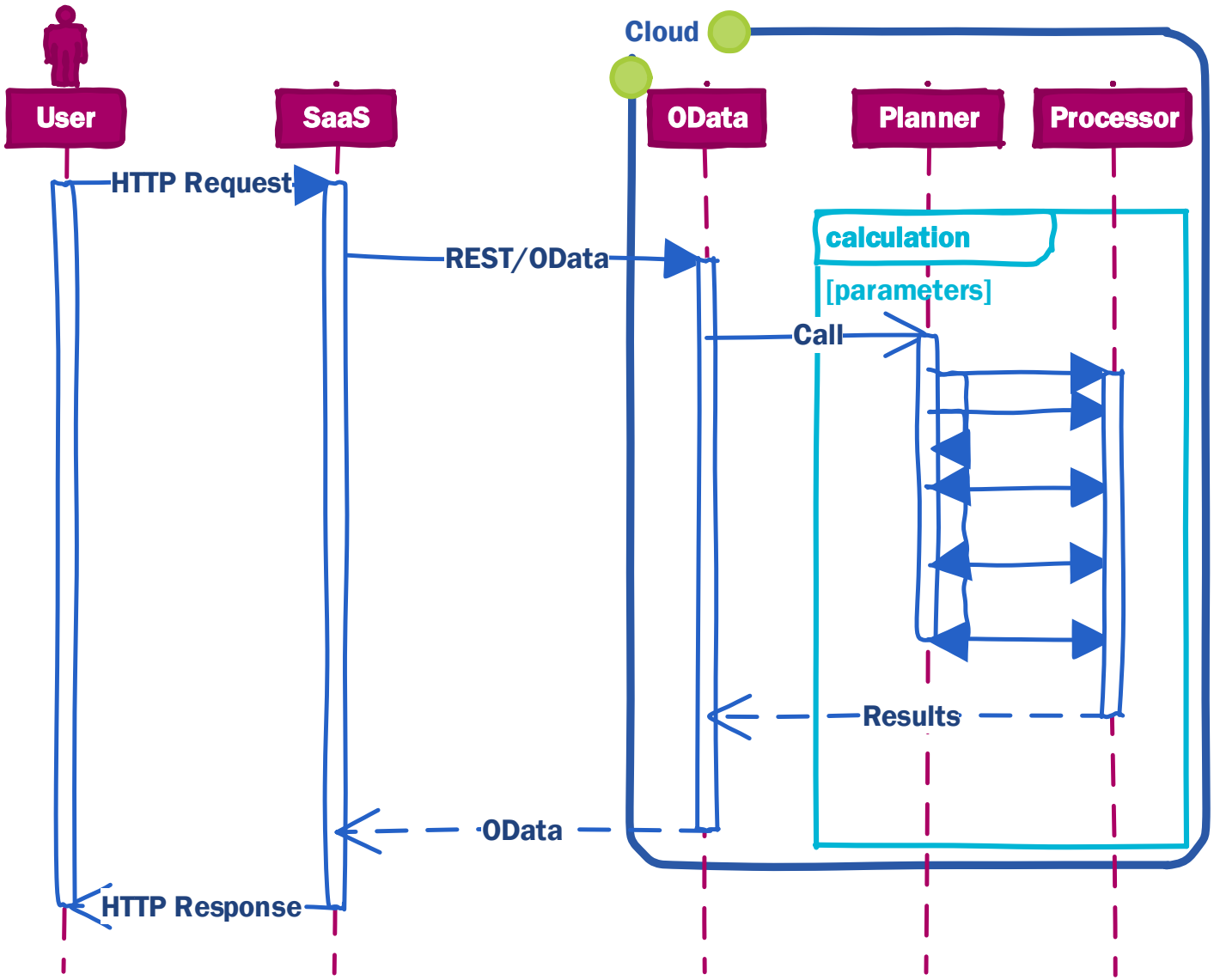
Core Solution: Components

- Operational Data – represents a composition of transactional data sources
- Calculation Engine – uses In-Memory Data Grid to performs Custom Calculation Code on a cluster
- In-Memory Data Grid – cluster environment such as GridGain Data Fabric or Gigaspaces XAP. Control all aspects of storing data and distributed calculations on a cluster
- Custom Calculation Code – implements various calculation algorithms
- Persistent Store – used to store Calculation Engine's operational data, when In-Memory Data Grid is off

Calculation Engine Functionality

- It performs one-path calculations, like:
  - Cumulative and Annualized performance Reports, Allocation Reports with different types of aggregation
  - Calculation of Alpha, Beta, Average Capital Base, TWR (Time-Weighted Return), Currency conversion
- It performs iterative calculations, like:
  - IRR (Internal Rate of Return) – requires to solve nonlinear equations (~2 ms per node)
- The OData v4 (http://www.odata.org) has been implemented based on open source Apache OLingo library (https://olingo.apache.org)

Typical Data Flow



OData: Binding Custom Code to Data Model

- Typical OData query to show TWR for last year:

```
$select=Name,TwrYtd&$filter=Date eq '2014-12-31' and CurrencyISO eq
'USD'
```

- In EDM file:

```
<field name="TwrYtd" type="Edm.Double">
   twr(args.setMonthOffset(-12).setAnnualized(true))</field>
<field name="Name" type="Edm.String">
   hrhy(args).name</field>
```

- MVEL expressions binds analytical functions to column definitions:
  - MVEL parser builds AST for columns and filter
  - Analytical functions request their dependencies
  - Planner builds dependencies graph

Divide and ~~Conquer~~ Synchronize: Parallel implementation of algorithms

- Dividing – need to be sure that these is the way to divide the whole job into independent pieces or to the pieces that at least could be executed in parallel to some extent:
  - Calculations on leaves:
    - They are usually independent and thus could be performed in parallel
  - Calculations and aggregation on nodes should either:
    - For additive calculations like TWR – wait for all underlying calculations
    - For semi-additive calculations like IRR – collect data needed to calculate – then could be done in parallel
- Synchronization – combine individual results into result of whole job and suspend execution of pieces until they get all their dependencies satisfied
  - `Callable` and `Future` have been used since Java 5
  - Recursive algorithms in parallel environment are potential victims of thread starvation

Thread starvation

- When number of nodes waiting for its prerequisites exceeds number of available threads in fixed size Thread pool, the application slows down
- There is a technique called “continuation”, which is devised to address thread starvation:
  - Start a child task (which will possibly produce a number of child jobs), get a task future
  - Detach this current job from a worker thread, allowing it to run other jobs
  - Register a completion listener on the task future, that will resume this current job (attach it back to a worker thread)
  - When resumed back - handle the child task result and return
- It is really hard to design, write the code and debug the continuation-based implementation of recursive algorithms

Non-blocking Parallel Recursive Calculations

Generalized approach uses the idea of topological sort where independent vertices have been removed. There are two moving parts:

- Planner, which builds graph of dependencies, which is very obvious task for tree-like structures and oriented graphs
- Processor, which is looking for independent leaves and nodes and submit them to the grid for execution

How it works

- Processor never uses `future.get()` method, it registers completion listener instead
- When listener get control, it checks for nodes, for which it would be the last dependence, and submit them for the execution
- When listener found the last node (root in tree-like structures) in dependency graph, it posts waiting thread with result of whole calculation
- When Processor can not find any submittable node, it means that it found a cyclic dependency

The City That Never Sleeps
The Grid That Never Waits

Dependency-driven execution of recursive calculation allows to:

- Avoid thread starvation
- Use external thread (webservices' thread in case of OData) for synchronization
- Make parallel implementation of algorithm much more straight-forward
- Simplify cache management for once calculated values

Memory-efficient Object Data Model

- Object are more expensive than a primitives, thus `Integer` takes 16 bytes memory to store its value, while `int` takes only 4 bytes
- Most of in-memory grids offer only one data structure to store in-memory data on a grid-cache, which is sharded map. You may significantly reduce memory footprint by re-organizing data structures from mechanically converted relational data structures to object model:
  - Structure for storing currency conversion table could be represented like this:
    - `GridCache<Integer, Pair<Date, Double>> currencies;`
  - Or like this, using implied array index as a day since 1 Jan 1904:
    - `GridCache<Integer, double[]> currencies;`
  - When you have less than a 10% values, you may use sparse arrays:
    - `GridCache<Integer, DoubleMatrix1D<Double>> currencies;`

Looking forward to use more open source

- The Goldman Sachs collection framework offers a galore of very memory-efficient collections (https://github.com/goldmansachs/gs-collections/wiki)

Lessons learned

In real-life you have to keep in mind:

- How to make calculations parallel
- Design of memory-efficient object data model
- Expandable data model on top objects for easy integration

Life insurance

Use case: Monte-Carlo simulation for life insurance

In life insurance industry companies use Monte-Carlo simulation to estimate the value of its policies

- Typically the process generates ~20 million (~50 cashflows * 360 months * 1000 Monte-Carlo paths) numbers for one policy, which gives for a million policies a 20 trillion numbers or 80TB of data
- It requires about a 20 minutes for a 2000 cores to generate such data
- Then data is grouped into ~500 cohorts, which gives 20,000,000 * 500 = 10 billion numbers or only 40GB of data
- The analytics and researches use that grouped data
- The MS SQL Server simply isn't capable to perform required grouping on such volume of data

Parallel computing and Monte-Carlo

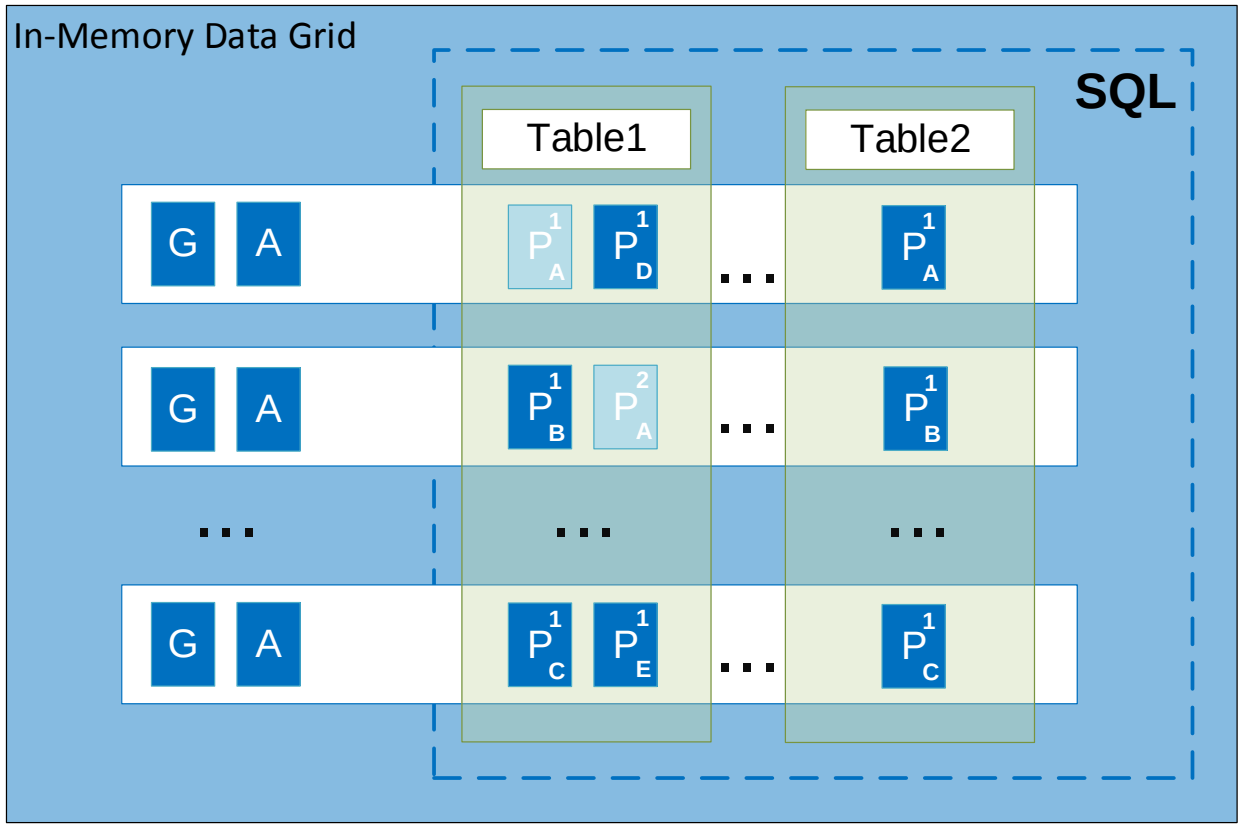
- In the financial sector, Monte Carlo method is used to assess the value of companies, perform risk analysis, or calculate financial derivatives. The method relies on repeated random sampling by running simulations multiple times in order to calculate the same probabilities heuristically.
- Monte Carlo Simulations is suited very well for In-Memory grids
- Problems that fit the Monte Carlo method are easily split into parallel execution
- Challenges are:
  - To avoid storing the whole 80TB of all generated Monte-Carlo paths
  - To provide faster generation of subset of Monte-Carlo paths for what-if scenarios

In-Memory Data Grid: The Solution

- Use In-Memory Data Grid to generate future cash-flows, aggregate them by cohorts, calculate conditional tail expectations, store and query results in same computer cluster where data are evenly distributed between cluster nodes, provides high performance, quick response and scale-out capability.
- The proposed solution utilizes mixed computational paradigm:
  - An Object mode, where objects are stored in distributed collections (maps, lists, multidimensional arrays) for complex iterative calculations and hierarchal roll-ups
  - A SQL mode for easy integration with report generators and BI tools, where collections of objects are treated as relational tables

Data Layout

- Policy distribution
- Cash-flow generation and:
  - intra-node aggregation
  - Inter-node aggregation
- Redistribution and Storing
- Querying and Analyzing



G – cash-flows generator

A – cash-flows aggregator

P^1_a partition A segment 1

In-Memory Data Grid: The Solution (cont.)

- Policy distribution - policies are partitioned to maximize data colocation, and to prevent data skew; partitions are dynamically divided into fixed size segments. This technique called partition overflow and it is a tradeoff between uniformity of distribution and aggregation complexity
- Intra-node aggregation – as soon as cash-flow generation is done, it get aggregated within the partition segment
- Inter-node aggregation - intermediate aggregated results from the segments are combined together within whole partition
- Redistribution and Storing - to make even distribution and thus effective distributed querying/calculations possible, data have to be split into relatively small pieces and stored on all nodes

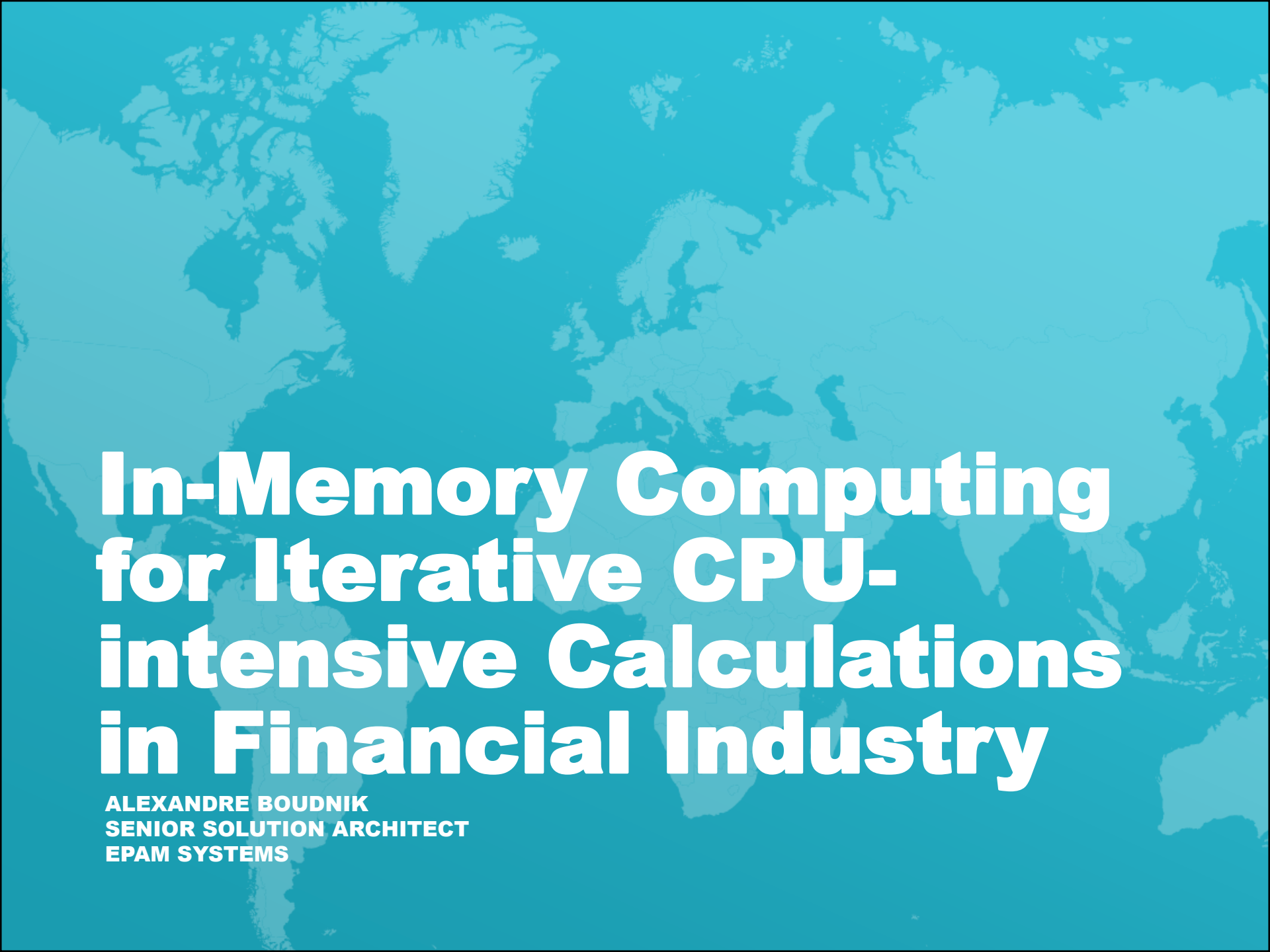
In-Memory Data Grid: The Solution (cont.)

- Querying and Analyzing
  - An Object mode, a Distributed Executor Service is used for complex iterative calculations and hierarchal roll-ups
  - In SQL Mode each node contains a subset of the data allowing to leverage a whole grid to process entire data set. Obviously, sequential scans are performed in parallel. Joins are executed in parallel with technics known as broadcasting and re-distribution
- Support for what-if scenarios
  - Depends on what user has changed in cashflow definition, the system can invalidate only affected partitions and avoid of costly recalculation of entire set of Monte-Carlo paths

Lessons learned

- Batch processing and core SQL doesn’t scale to full blown simulations\
- Hadoop stack doesn’t fit near real-time expectations even with moderate data volumes
- If brute-force doesn’t work – don’t use it. At all
- Replace horizontal scaling with creative re-thinking of data models and processes organization

Q & A



In-Memory Computing for Iterative CPU-intensive Calculations in Financial Industry

ALEXANDRE BOUDNIK
SENIOR SOLUTION ARCHITECT
EPAM SYSTEMS