Public

SAP HANA: A Data Platform for Enterprise Applications Purpose Built for Modern Hardware

Anil K Goel, SAP Canada

May 1, 2019

CMU 15-721

Agenda

- Recap: Traditional Enterprise Data Management Architectures
- Vision: Enterprise Operational Analytics Data Management Systems
- Reality: Design and Implementation Aspects of SAP HANA
  - Key Design Principles and Concepts
  - Focus: The new HANA EXecution Engine (HEX)
  - Focus: Storage Extensions
- Conclusion

About SAP

SAP is the **world leader in enterprise applications** in terms of software and software-related service revenue. Based on market capitalization, we are the world's **third largest independent software manufacturer**.

437,000+
Customers in more than 180 countries

98,500+
Employees in 144+ countries (03-31-2019)

€24.74bn
Total Revenue (non-IFRS) in FY2018

92%
Of Forbes Global 200 are SAP customers

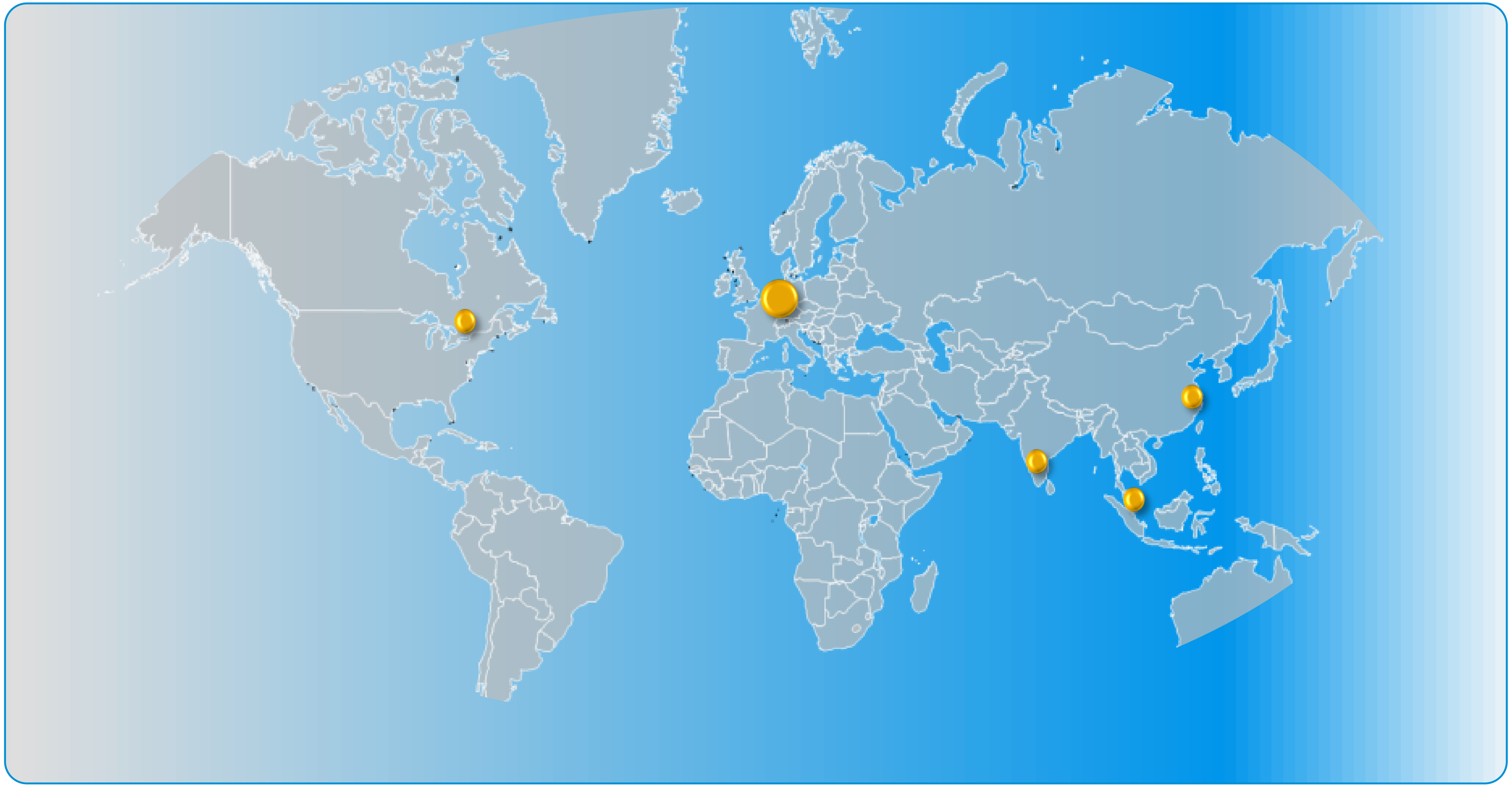
47 yrs.
Of history and innovation

100+
Innovation and development centers

19,200+
SAP partner companies globally

~195 mil.
Users in SAP cloud user base

Global Database Development Team



SAP Labs Waterloo

Established Database R&D site going back 35+ years

~220 people

- Majority SAP Database Research and Development
- Work on HANA, Cloud-native databases, distributed databases, Edge Computing

Co-op & Grad Student intern programs

Strong academic collaborations

We are hiring!

- Full-time positions
- Student jobs - master/bachelor theses, graduate internships

http://www.careersatsap.com/

http://jobs.sap.com

https://www.saphana.com

Traditional Enterprise Data Management Architectures

Transactional System for Online Transaction Processing (OLTP)

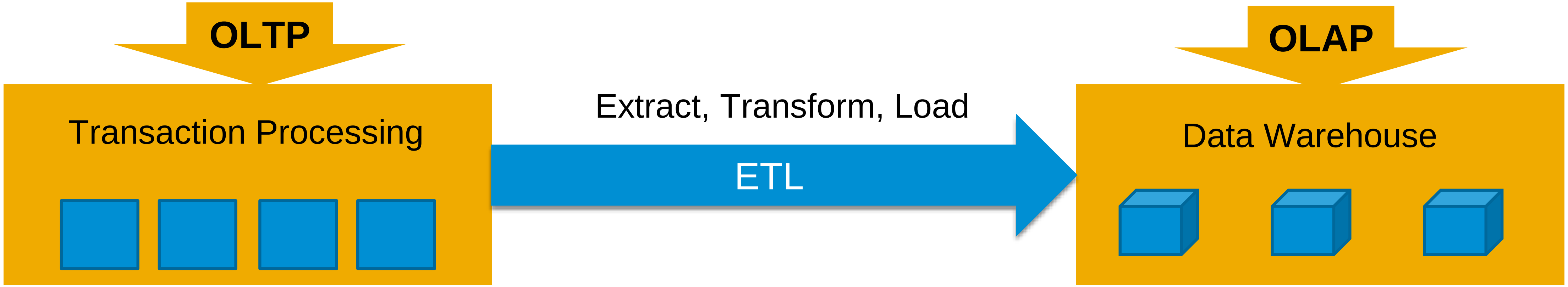
- Short-running, trivial statements; frequent updates; high amount of concurrent users

Data Warehouse for Online Analytical Processing (OLAP)

- Long-running, complex statements; (almost) read-only; few users

Nightly Data Extraction, Transfer, and Loading Between Systems (ETL)

- Transformation from OLTP-friendly to OLAP-friendly format; indexing; pre-aggregation



Challenges of Traditional Architectures

High Costs / TCO

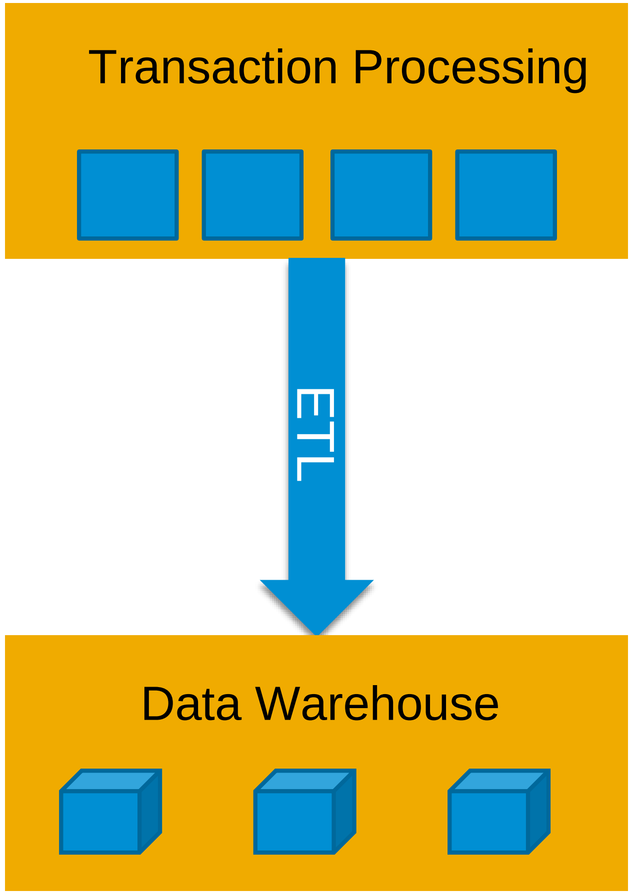
- Two (different?) database management systems
- Two times the license costs
- Two times the hardware
- Two times the administration

No Data Freshness for Reporting

- Data is updated only nightly
- Reporting during the day always sees stale data

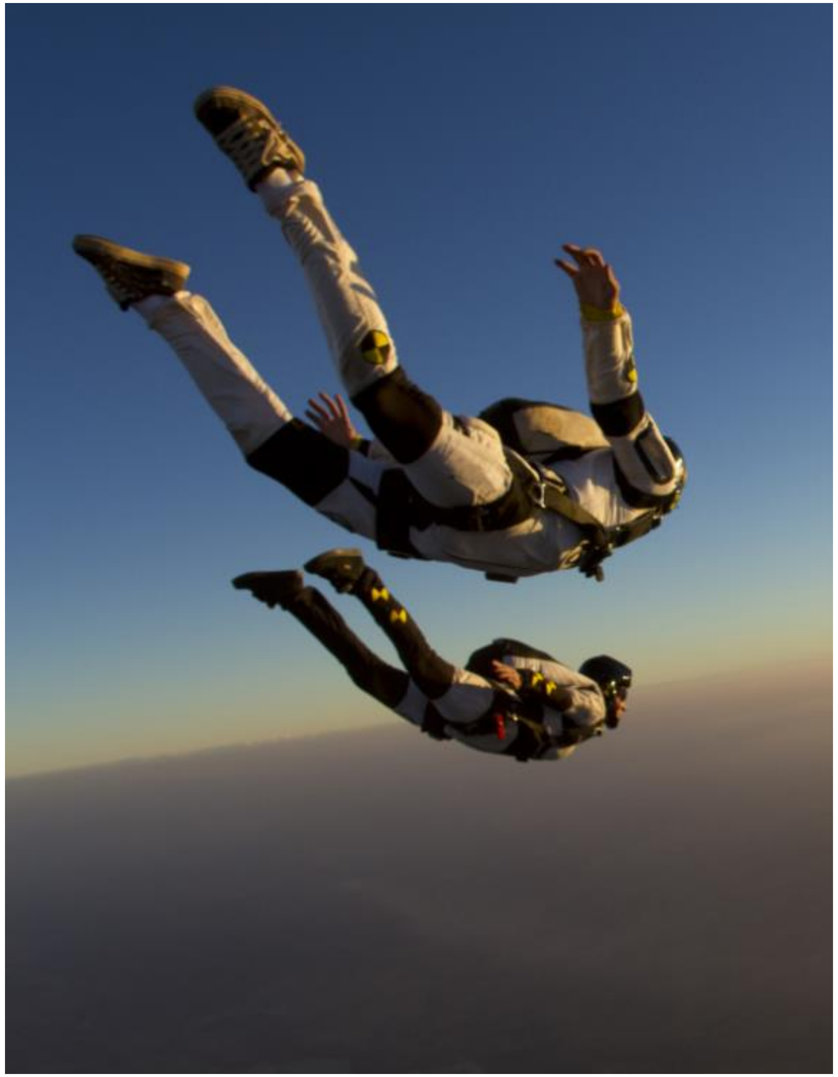
ETL Processes are a Pain

- Complex to design and maintain
- Need to complete in daily maintenance window (e.g. <8 hours)
- Data preparation necessary for reporting
- Slow reaction to changing business demands (no ad-hoc reporting)



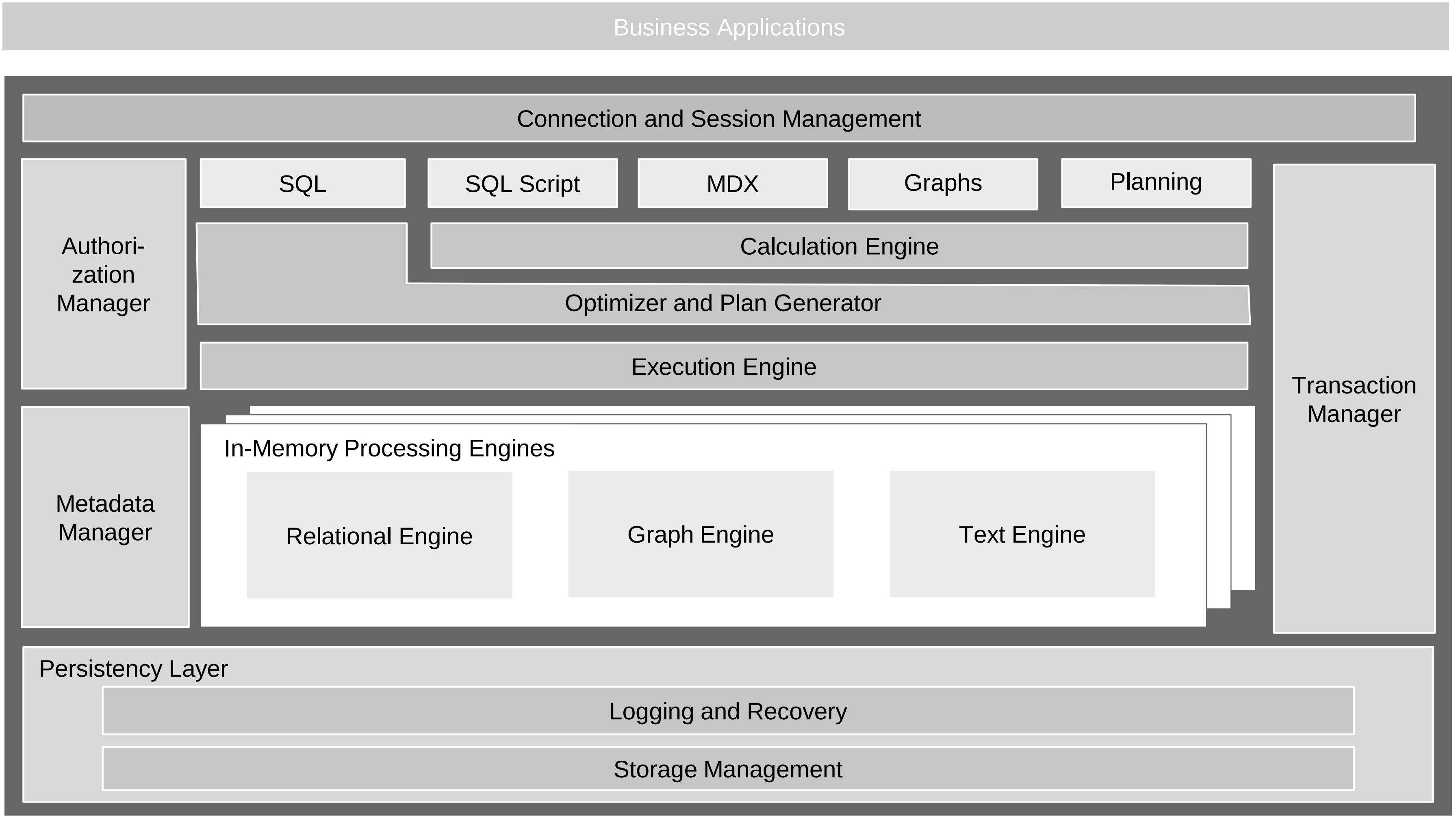
From Vision To Reality: Objectives for HANA

- Good Enough for OLTP
  - Unrealistic to beat a specialized, pure OLTP engine
  - But: Be able to sustain a typical enterprise workload
  - Example: 40.000 SQL statements / second
- Excel in Analytics
  - Flexible reporting without pre-computation / aggregates
  - Leverage modern hardware (multicore + large DRAM capacity)
  - Marketing: „Subsecond everything“
  - Vision: „window of opportunity“
- Support both OLTP and OLAP in ONE database

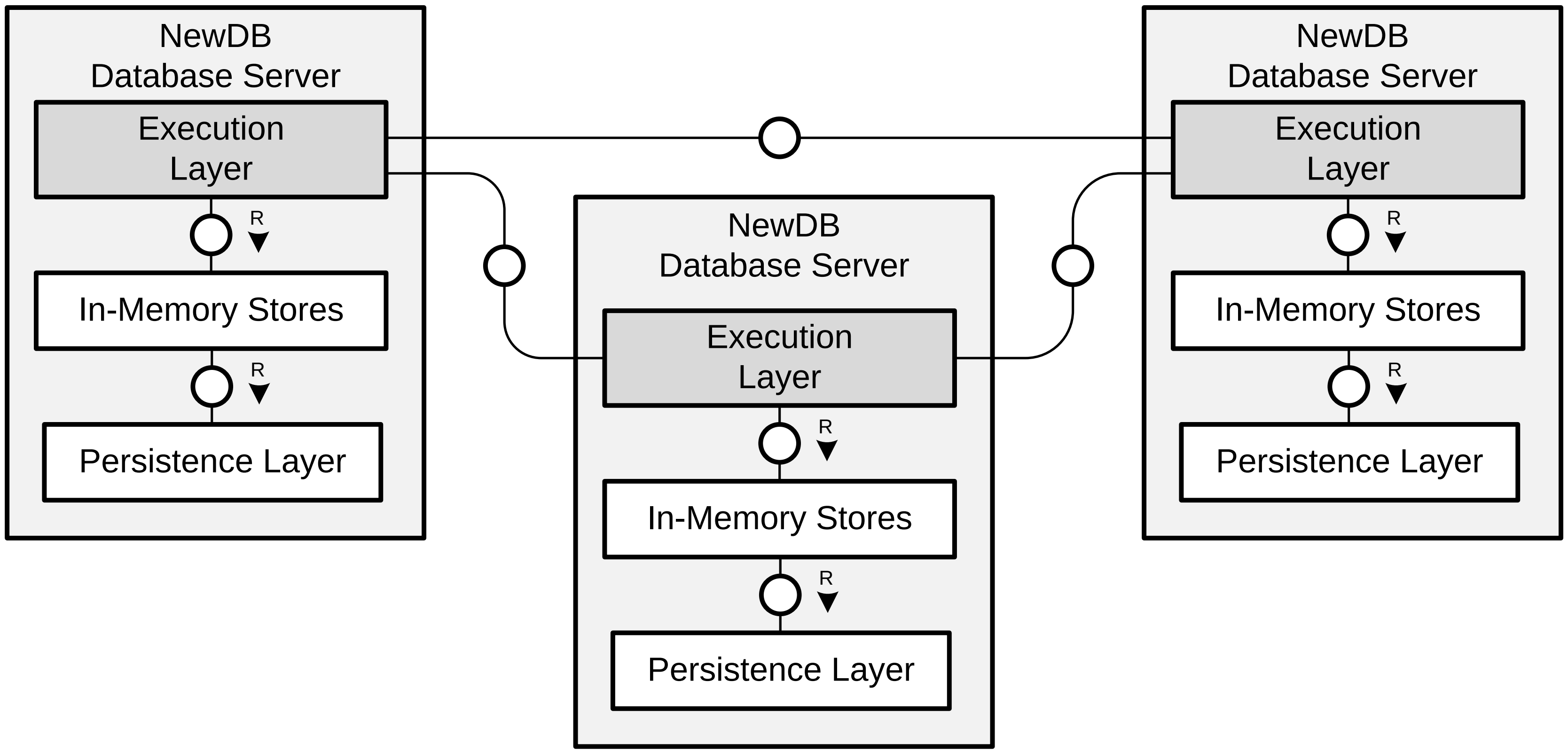


Key Design and Implementation Aspects of SAP HANA

SAP HANA Database
Multi-Engine for Multimodal Enterprise Applications

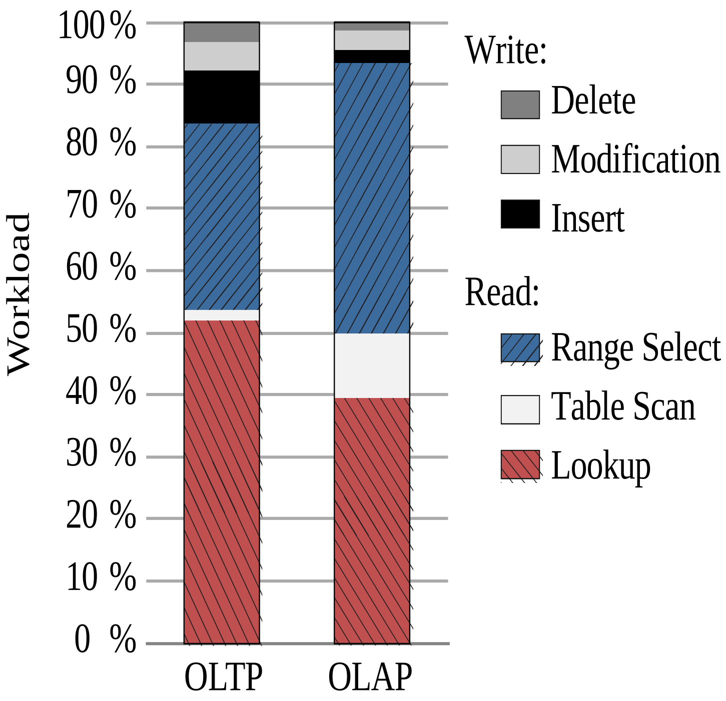


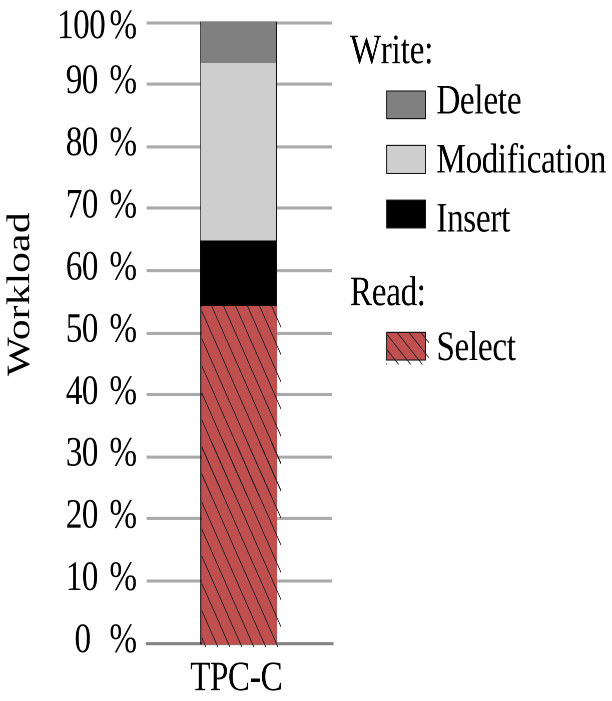
Distributed Share-Nothing In-Memory Computing



Observation: Enterprise Workloads are Read Dominated

- Workload in Enterprise Applications consists of:
  - Mainly read queries (OLTP 83%, OLAP 94%)
  - Many queries access large sets of data





SAP HANA Column Store Main: Fast Queries

- Read-Optimized, Immutable Data Store
- Dictionary Compression
  - All data in columnar tables is dictionary compressed
  - Dictionary is prefix-compressed
  - Dictionary is sorted in same order as data values
- Efficient secondary data compression (run-length, cluster, prefix, etc.)
  – Heuristic algorithm orders data to maximize compression of columns
- Compression schemes work well, e.g.,
  – Speeding up operations on columns to factor 10
  – Reduces Storage up to factor 5 for typical SAP data schemes

Dictionary

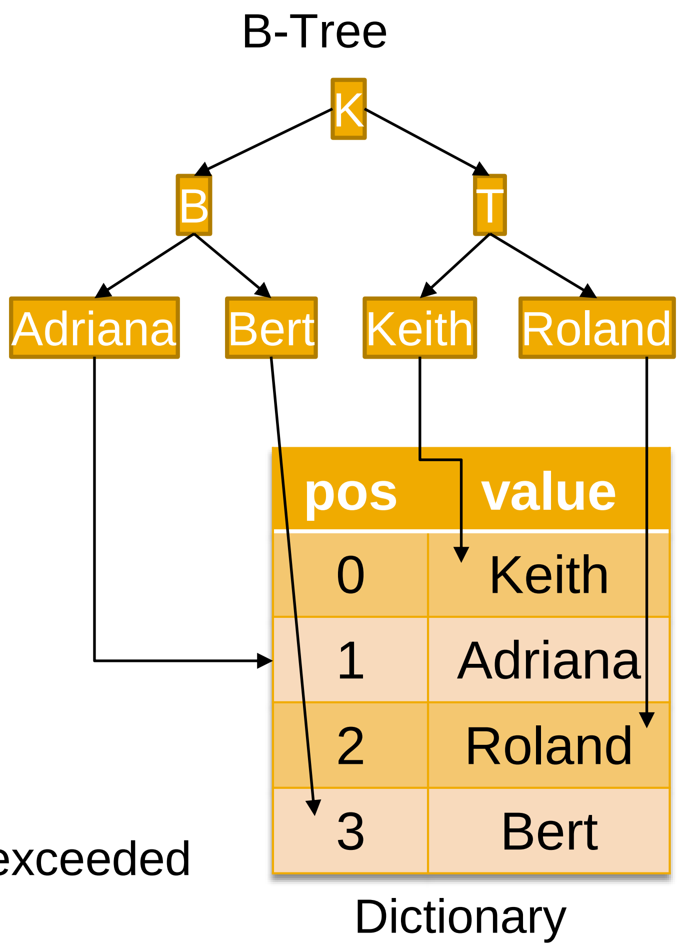
pos	value
0	Adam
1	Adriana
2	Alexa
3	Amber

Indexvector

pos	valueID
0	0
1	0
2	0
3	1
4	0
5	0
6	2
7	3

SAP HANA Column Store Delta: Write Optimized Update Support

- Write-Enabled Table Fragments Handle all Updates
  - Only update operation on main is to delete rows
  - UPDATEs modelled as DELETE+INSERT
  - INSERT append to delta store
- Dictionary not Sorted
  - No need to recode column vectors upon delete/insert
- Additional B-Tree for Efficient Lookup
  - Allows to quickly retrieve valueID for value
  - Essential for fast unique checks upon insert
  - Can be used for range queries
- Less compression of data
- Delta is merged with main periodically, or when thresholds exceeded
  - Delta merge for a table partition is done on-line, in background
  - Enables highly efficient scan of Main again

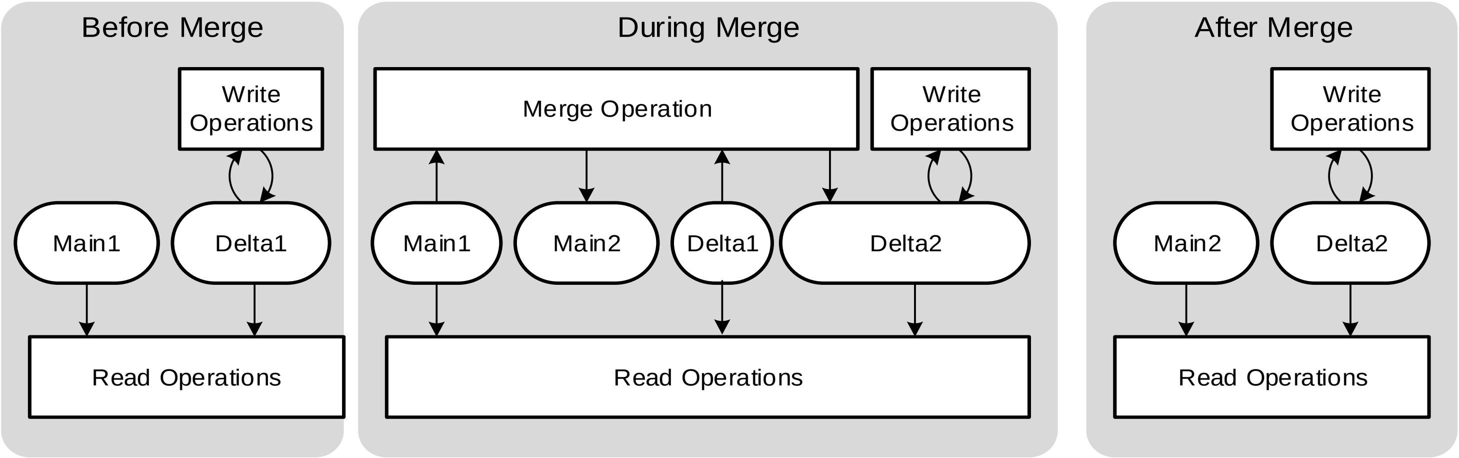


Indexvector

pos	valueID
0	0
1	0
2	0
3	1
4	0
5	0
6	2
7	3

Delta Merge

- Consolidation of Delta and Main into new Main
  - Improves query performance (especially for analytics)
  - Reduces memory footprint (no B-Tree for dictionary necessary)
- Automatically triggered by the System based on Cost-Based Decision Function
  - Considers delta:main ratio, size in RAM and disk, system workload
  - Performed on a per table-basis (actually: partition-based), parallelized on column-level



SAP HANA Technology
Compression with run length encoding

Classical Row Store
Difficult to compress

Company [CHAR50]	Region [CHAR30]	Group [CHAR5]
INTEL	USA	A
Siemens	Europe	B
Siemens	Europe	C
SAP	Europe	A
SAP	Europe	A
IBM	USA	A

HANA Column Store:
Dictionary compressed

0 INTEL 1 Siemens 2 SAP 3 IBM	0 Europe 1 USA	0 A 1 B 2 C
0	1	0
1	0	1
1	0	2
2	0	0
2	0	0
3	1	0

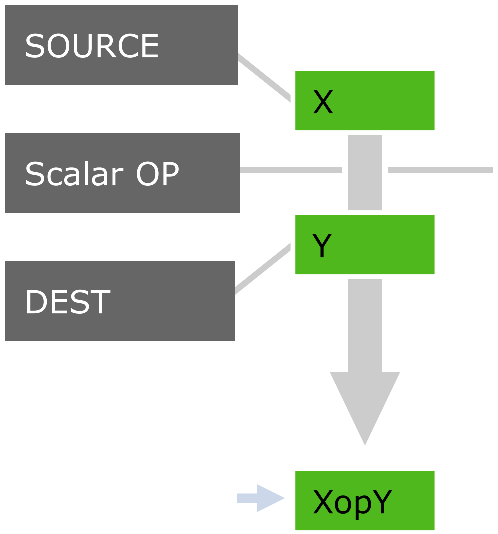
HANA Column Store:
Run length compressed*

0 INTEL 1 Siemens 2 SAP 3 IBM	0 Europe 1 USA	0 A 1 B 2 C
1 x „0“	1 x „1“	1 x „0“
2 x „1“	4 x „0“	1 x „1“
2 x „2“	1 x „1“	1 x „2“
1 x „3“		3 x „0“

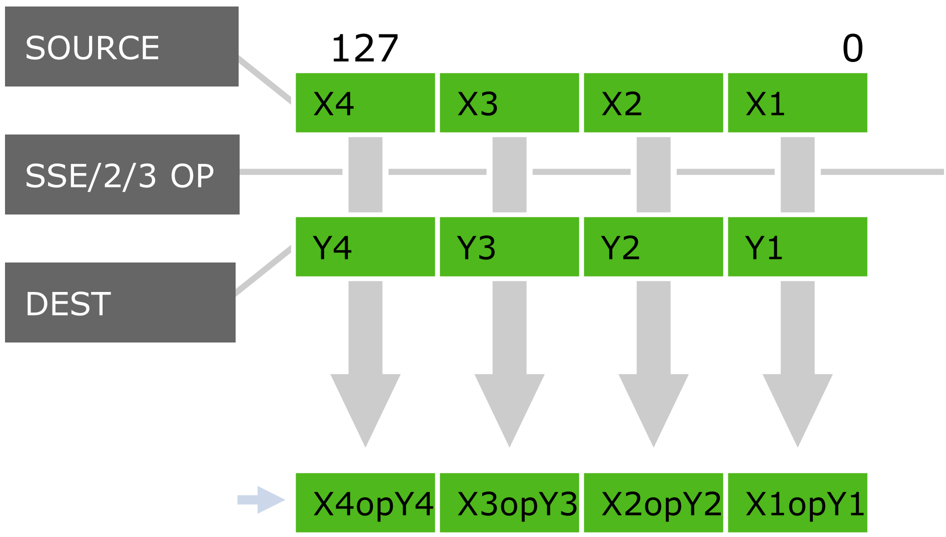
* Note that there is a variety of compression methods and algorithms like run-length compression

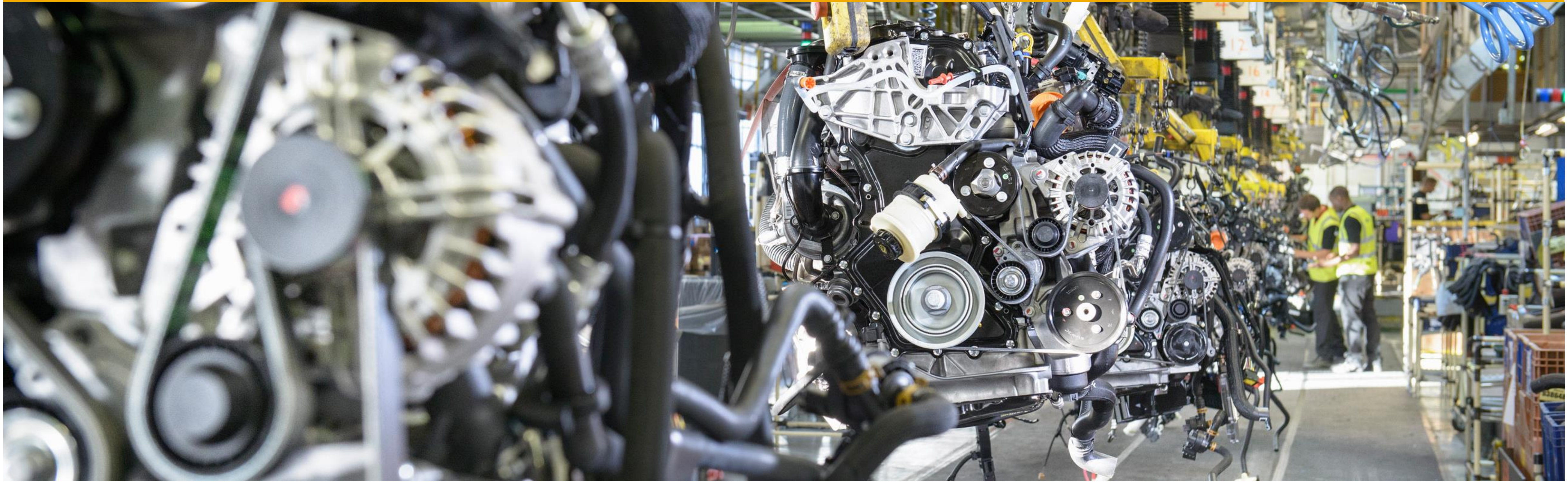
Single Instruction Multiple Data (SIMD)

Scalar processing
- traditional mode
- one instruction produces one result



SIMD processing
- with Intel® SSE & AVX
- one instruction produces multiple results



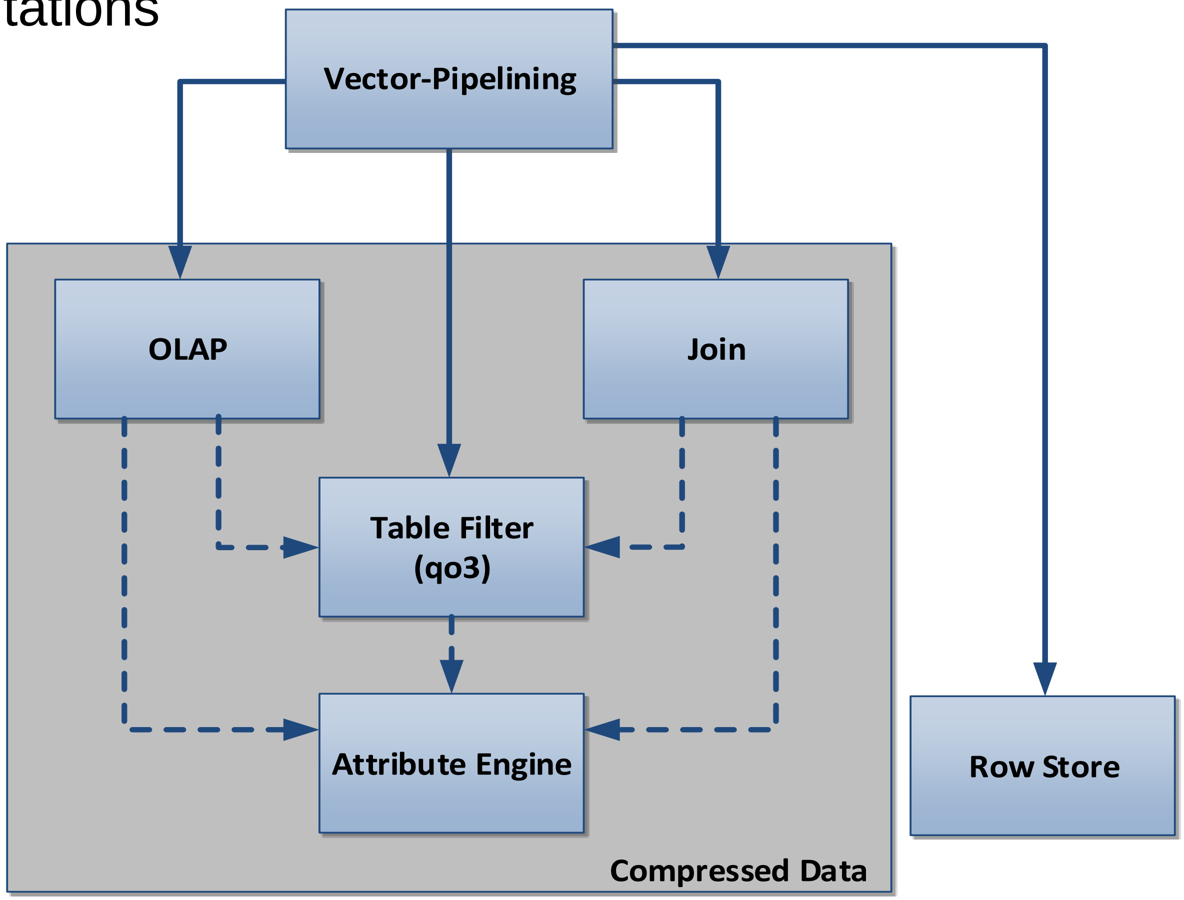


Technical Deep Dive

The New HANA EXecution Engine (HEX)

Status Quo: HANA Query Processing Engines

- Operate on dictionary-compressed data representations
- Designed with analytics in mind
  - Also: Optimized and tuned for OLTP
  - Example: Full semi-join reduction for all joins



HANA as a general purpose database: Novel focus areas

- Already excellent performance
- New hot spot: Short-running queries
  - Allow migration of existing / legacy application **without code changes**
  - Many applications are not DBMS friendly
- New hot spot: Memory footprint reduction
  - HANA materializes intermediate results between operators
  - Good strategy for analytics, but high intermediate memory consumption
  - Alternative: Pipelined execution

General Purpose Database

Next generation query processing in HANA

- Goal: Extend HANA with a next-generation execution engine
  - Reduce runtime memory footprint
  - Replace „old“ engines to avoid complexity increase
- Maintain competitive advantage
  - Exploit columnar and dictionary-compressed storage
  - Operate on compressed data representations
- Incorporate latest research results: Code generation
  - Academic frontrunner: TU Munich / HyPer
  - Commercial systems: SAP HANA Vora/Velocity, Microsoft Hekaton

Efficiently Compiling Efficient Query Plans for Modern Hardware

Thomas Neumann
Technische Universität München
Munich, Germany
neumann@in.tum.de

ABSTRACT

As main memory grows, query performance is more and more determined by the raw CPU costs of query processing itself. The classical iterator style query processing technique is very simple and flexible, but shows poor performance on modern CPUs due to lack of locality and frequent instruction mispredictions. Several techniques like batch oriented processing or vectorized tuple processing have been proposed in the past to improve this situation, but even these techniques are frequently out-performed by hand-written execution plans.

In this work we present a novel compilation strategy that translates a query into compact and efficient machine code using the LLVM compiler framework. By aiming at good code and data locality and predictable branch layout the resulting code frequently rivals the performance of hand-written C++ code. We integrated these techniques into the HyPer main memory database system and show that this results in excellent query performance while requiring only modest compilation time.

1. INTRODUCTION

Most database systems translate a given query into an expression in a (physical) algebra, and then start evaluating this algebraic expression to produce the query result. The traditional way to execute these algebraic plans is the iterator model [8], sometimes also called Volcano-style processing [4]: Every physical algebraic operator conceptually produces a tuple stream from its input, and allows for iterating over this tuple stream by repeatedly calling the *next* function of the operator.

This is a very nice and simple interface, and allows for easy combination of arbitrary operators, but it clearly comes from a time when query processing was dominated by I/O and CPU consumption was less important: First, the *next* function will be called for every single tuple produced as intermediate or final result, i.e., millions of times. Second, the call to *next* is usually a virtual call or a call via a function pointer. Consequently, the call is even more expensive than a regular call and degrades the branch prediction of modern

Proceedings of the VLDB Endowment, Vol. 4, No. 9

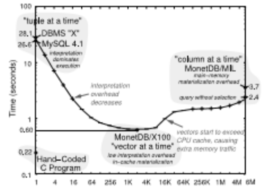


Figure 1: Hand-written code vs. execution engines for TPC-H Query 1 (Figure 3 of [16])

CPUs. Third, this model often results in poor code locality and complex book-keeping. This can be seen by considering a simple table scan over a compressed relation. As the tuples must be produced one at a time, the table scan operator has to remember where in the compressed stream the current tuple is and jump to the corresponding decompression code when asked for the next tuple.

These observations have led some modern systems to a departure from this pure iterator model, either internally (e.g., by internally decompressing a number of tuples at once and then only iterating over the decompressed data), or externally by producing more than one tuple during each *next* call [11] or even producing all tuples at once [1]. This block-oriented processing amortizes the costs of calling another operator over the large number of produced tuples, such that the invocation costs become negligible. However, it also eliminates a major strength of the iterator model, namely the ability to *pipeline* data. Pipelining means that an operator can pass data to its parent operator without copying or otherwise materializing the data. Selections, for example, are pipelining operators, as they only pass tuples around without modifying them. But also more complex operators like joins can be pipelined, at least on one of their input sides. When producing more than one tuple during a call this pure pipelining usually cannot be used any more, as the tuples have to be materialized somewhere to be accessible. This materialization has other advantages like allowing for vectorized operations [2], but in general the lack of pipelining is very unfortunate as it consumes more memory bandwidth.

An interesting observation in this context is that a hand-written program clearly outperforms even very fast vectorized systems, as shown in Figure 1 (originally from [16]). In a way that is to be expected, of course, as a human might use tricks that database management systems would never come

Code generation in HEX

- HEX uses LLVM for machine code generation
- Does not generate LLVM intermediate representation (IR) directly
- Uses SAP‘s „L“ language frontend instead
  - L is already used within HANA, e.g. for stored procedures
  - Code is easier to read than IR
  - Benefit from existing infrastructure (supportability, debugging, profiling…)
  - L already contains implementations of SQL expression functions (add_years(), …)

The Dark Side of code generation

Compilation Times

- To mitigate, HEX makes use of the Interpreter provided by L
  - Each piece of code is executed with the interpreter at first
    => Heavy performance penality, but execution can start immediately
  - When executed often: trigger compilation via LLVM (asynchronously)
  - Once finished, execution switches to the compiled version

- L interpreter is not optimal yet
  - Interprets L program (instructions)
  - Optimized bytecode interpreter provides much better performance
  - See recent work from TU Munich

2018 IEEE 34th International Conference on Data Engineering

Adaptive Execution of Compiled Queries

André Kohn, Viktor Leis, Thomas Neumann
Technische Universität München
{kohna,leis,neumann}@in.tum.de

***Abstract*—Compiling queries to machine code is a very efficient way for executing queries. One often overlooked problem with compilation is the time it takes to generate machine code. Even with fast compilation frameworks like LLVM, generating machine code for complex queries often takes hundreds of milliseconds. Such durations can be a major disadvantage for workloads that execute many complex, but quick queries. To solve this problem, we propose an adaptive execution framework, which dynamically switches from interpretation to compilation. We also propose a fast bytecode interpreter for LLVM, which can execute queries without costly translation to machine code and dramatically reduces the query latency. Adaptive execution is fine-grained, and can execute code paths of the same query using different execution modes. Our evaluation shows that this approach achieves optimal performance in a wide variety of settings—low latency for small data sets and maximum throughput for large data sizes.**

I. Introduction

Compiling queries to machine code has become a very popular method for executing queries. Compilation is used by a large and growing number of commercial systems (e.g., Hekaton [1], [2], MemSQL [3], Spark [4], and Impala [5]) as well as research projects (e.g., HIQUE [6], HyPer [7], DBToaster [8], Tupleware [9], [10], LegoBase [11], ViDa [12], Vodoo [13], Weld [14], Peloton [15], [16]). The main advantage of compilation is, of course, efficiency. By generating code for a given query, compilation avoids the interpretation overhead of traditional execution engines and thereby achieves much higher performance.

One obvious drawback of generating machine code is compilation time. Consider, for example, the following meta data query:

```
SELECT c.oid, c.relname, n.nspname
FROM pg_inherits i
JOIN pg_class c ON c.oid = i.inhparent
JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE i.inhrelid = 16490 ORDER BY inhseqno
```

This query touches only a very small number of tuples, which means that its execution time is negligible (less than 1 millisecond in HyPer). However, before HyPer can execute this query, it needs to compile it to machine code. With optimizations enabled, LLVM takes 54ms to compile this query. In other words, compilation takes 50 times longer than execution. Assuming a workload where similar queries are executed frequently, 98% of the time will be wasted on compilation. And this query is still fairly small; compilation times can be much higher for larger queries. Compilation of the largest TPC-DS query, for example, takes close to 1 second. Of course, for large data sizes, compilation does pay off as the resulting code is much more efficient than interpretation.

In this work, we focus on database systems that compile queries to LLVM IR ("Intermediate Representation"), which is afterwards compiled to machine code by the LLVM compiler backend. This approach offers the same machine code quality as compiling to C/C++, while reducing compilation time by an order of magnitude [7]. The compilation times of the LLVM compiler may be low enough for some workloads, for example those consisting of long-running ad hoc queries or pre-compiled stored procedures. For other applications, however, long compilation times are a major problem.

The example query shown above is one of the queries sent by the PostgreSQL administration tool *pgAdmin*. On startup, pgAdmin sends dozens of complex queries (up to 22 joins), all of which access only very small meta data tables. Compiling these queries causes perceptible and unnecessary delays. Caching the machine code (e.g., after stripping out constants) might improve subsequent executions, but would not improve the initial user experience. More generally, because the human perception threshold is less than a second, the additional latency caused by compilation can lead to a worse user experience for interactive applications. Finally, business intelligence tools occasionally generate extremely large queries (e.g., 1 MB of SQL text), which de facto cannot be compiled with standard compilers.

For the workloads just mentioned, the user experience of a compilation-based engine can be *worse* than that of a traditional, interpretation-based engine (e.g., Volcano-style execution). Thus, depending on the query, one would sometimes prefer to have a compilation-based engine and sometimes an interpretation-based engine. Implementing two query engines in the same system, however, would involve disproportionate efforts and may cause subtle bugs due to minor semantic differences. In this work, we instead propose an adaptive execution framework that is principally based on a single compilation-based query engine, yet integrates interpretation techniques that reduce the query latency. The key components of our design are a (i) fast bytecode interpreter specialized for database queries, (ii) a method of accurately tracking query progress, and (iii) a way to dynamically switch between interpretation and compilation. Without relying on the notoriously inaccurate cost estimates of query optimizers, this dynamic approach enables the best of both worlds: Low latency for small queries and high throughput for long-running queries.

Our adaptive execution framework is directly applicable to many compilation-based systems. Furthermore, our approach

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00027

Additional challenges

- Materialization strategies (early vs late)
  - Currently: Late materialization to keep intermediate results small
  - Research: Always best approach?
- Tuple-at-a-time vs blockwise processing (goal: exploit SIMD instructions in complete pipeline)
  - Trade-off with register locality of tuples
- Finding the right mix of compilation / interpretation for very complex query plans
  - Research results (TUM): Compilation does not finish for extremely complex, generated queries
- …

Technical Deep Dive

HANA Native Storage Extensions: Adding disk processing to an in-memory database server

HANA Page Loadable Columns (SIGMOD - 2016)

Idea: load pieces of a column

- –Not every column of each table needs to be in memory
- –Not all rows of a column need to be in memory
- –Not all data structures of a column need to be in memory
  - Load portions of data vectors needed for a query
  - Load portions of dictionaries needed for search
  - Load portions of inverted indices corresponding to query predicates

Page in/out read optimized portion:

- More memory consumed by **read only** portion of a column

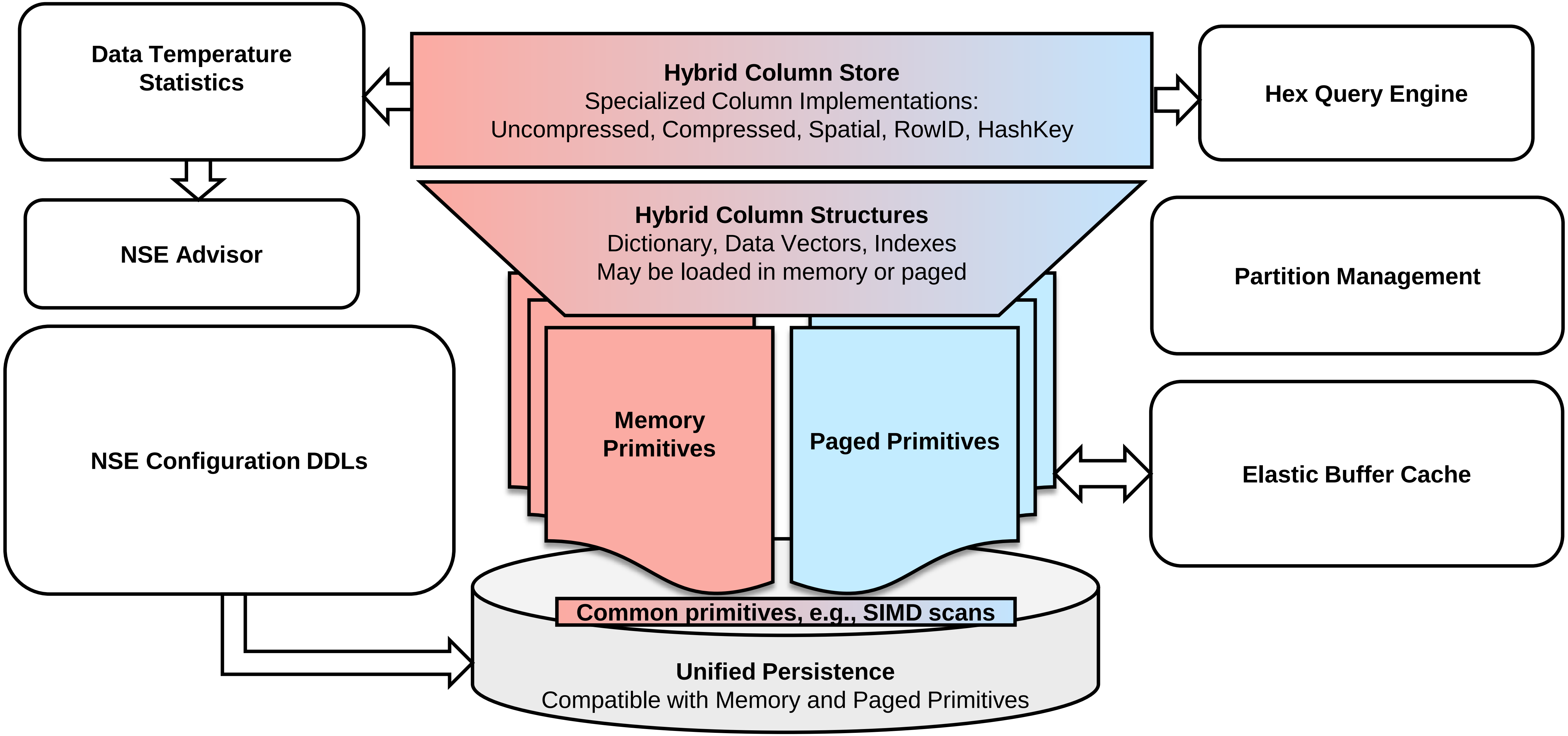
Native Storage Extension (VLDB 2019)

Value proposition:

- Increase HANA data capacity at low TCO
- Deeply integrated warm data tier, with full HANA functionality
- Support for all HANA data types and data models
- Simple system landscape
- Scalable with good performance
- Supported for both HANA on-premise and HANA-as-a-Service (HaaS)
- Available for any HANA application
- Complements, without replacing, other warm data tiering solutions (extension nodes, dynamic tiering)

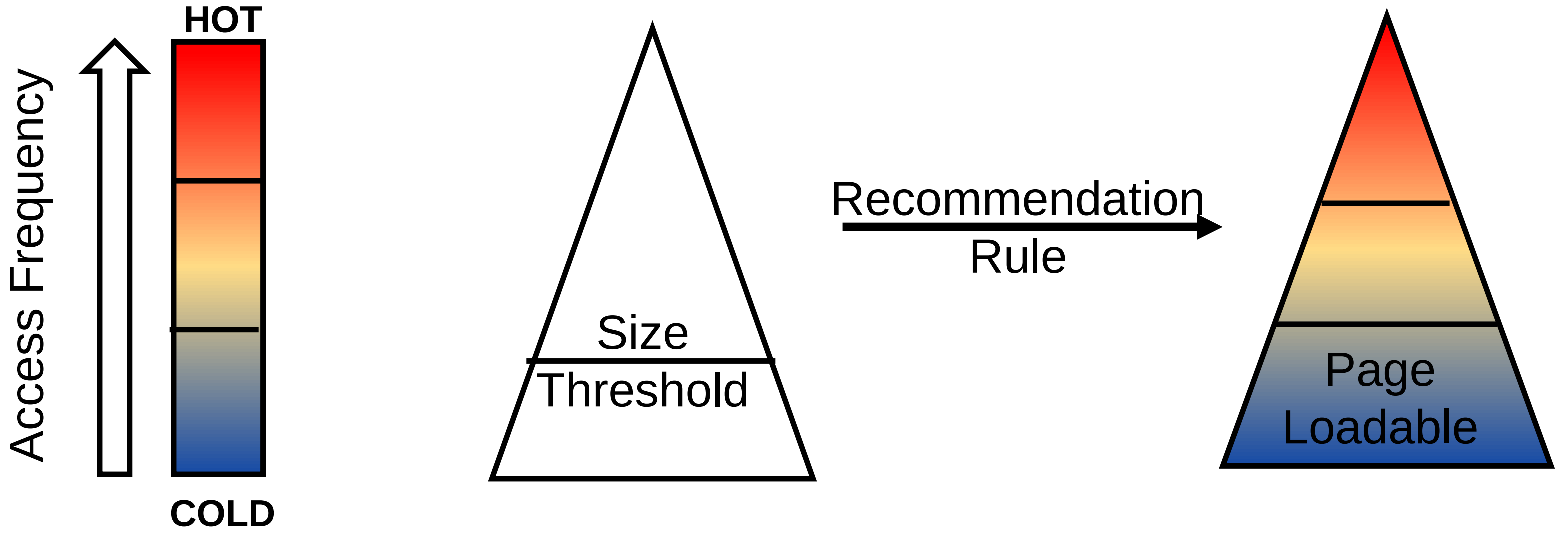
Use cases:

- Any customer built or SAP built HANA application that is challenged by growing data volumes
- S/4HANA and Suite on HANA data aging framework (replace current “page loadable columns” solution)
- Reduced storage size and TCO for cloud deployments of internal systems and HaaS applications
- Expand capacity in HANA extension nodes (BW use case being evaluated)


Data Temperature Statistics
Hybrid Column Store
Specialized Column Implementations:
Uncompressed, Compressed, Spatial, RowID, HashKey
Hex Query Engine
NSE Advisor
Hybrid Column Structures
Dictionary, Data Vectors, Indexes
May be loaded in memory or paged
Partition Management
Memory Primitives
Paged Primitives
NSE Configuration DDLs
Elastic Buffer Cache
Common primitives, e.g., SIMD scans
Unified Persistence
Compatible with Memory and Paged Primitives

NSE Recommendation Engine

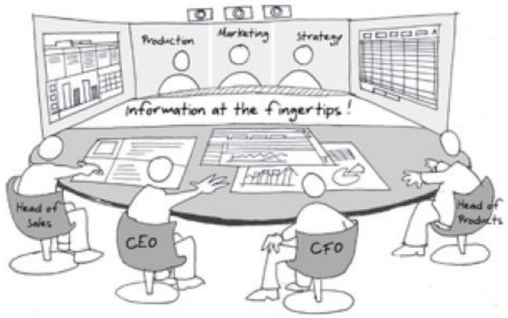
- Data access frequency collected at query level for each column (scan count etc.) to indicate data hotness
- Rule based heuristics to identify cold objects (column, partition, table) with large memory footprint as page loadable candidates
- User adjustable threshold for cold and hot objects
- Flexible interface for user to act on individual recommended object to convert from column loadable to page loadable or vice versa.



Wrapping up

Conclusion

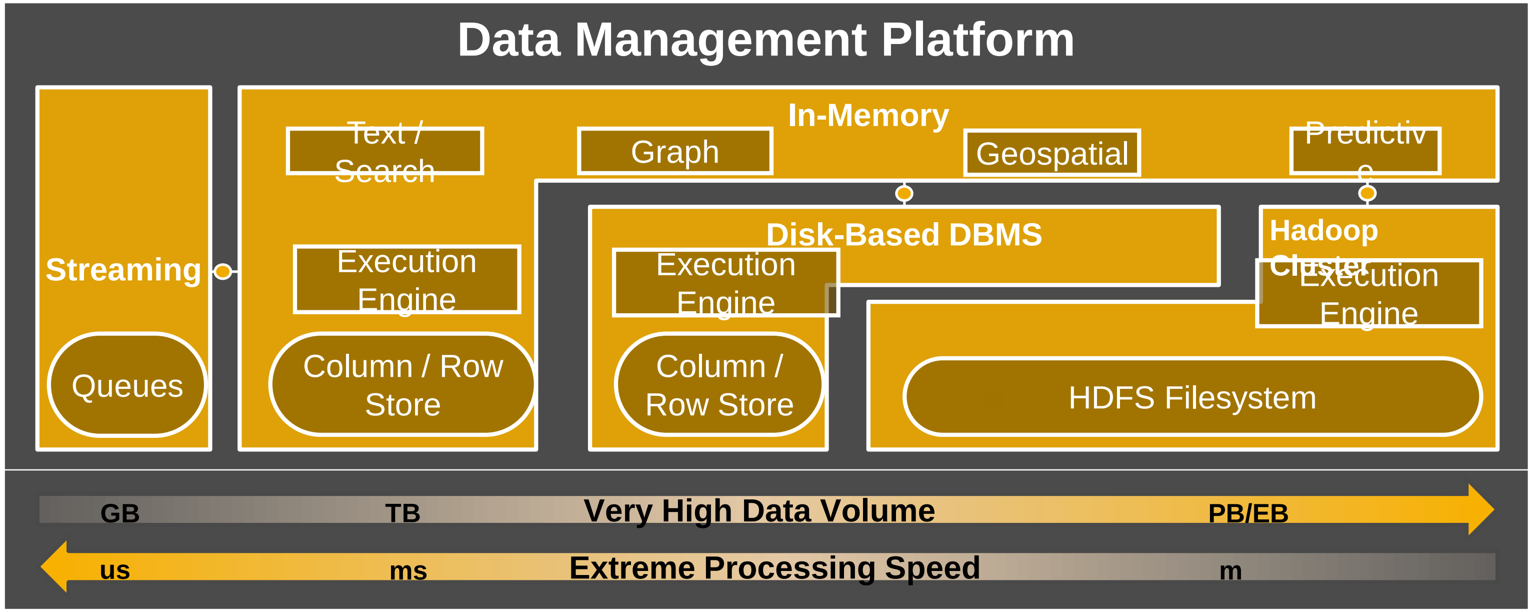
- Operational Analytics DBMS are a Game-Changer for Enterprise Applications
  - Fast reporting even without indexes / aggregates / materialized views (CRM)
  - Additional insights from reporting on transactional data (HANA Live / Suite on HANA)
  - Complete application redesign (S4HANA)
    - No aggregates (simplification)
    - Code pushdown
    - Fast analytics on current data in real-time



Challenges and Next Steps

Cloud, Memory Footprint, and Hardware Costs
Performance Challenged by Specialized Engines and NoSQL
Towards Polyglot Persistency / Data Management Platforms

CMU PDL Tech Talk: May 2, 2019, 12:00pm
https://db.cs.cmu.edu/events/spring-2019-anil-goel-sap/



Anil Goal, SAP Canada

Anil.Goel@sap.com

Run Simple