SQL Server Performance

SQL 2016 new innovations

Ivan Kosyakov

Technical Architect, Ph.D., http://biz-excellence.com

Microsoft Technology Center, New York

Mission-critical performance

Performance

In-Memory OLTP enhancements
Greater T-SQL surface area, terabytes of memory supported, and higher number of parallel CPUs

Operational Analytics
Insights on operational data; works with In-Memory OLTP and disk-based OLTP

Query Store
Monitored, optimized query plans

Temporal Tables
Query data as points in time and recover from accidental data changes and application errors

Security

Always Encrypted
Sensitive data remains encrypted at all times, with ability to query

Dynamic Data Masking
Real-time obfuscation of data to prevent unauthorized access

Row-Level Security
Fine-grained access control for table rows

Other enhancements
Audit success/failure of database operations

TDE support for storage of In-Memory OLTP tables

Enhanced auditing for OLTP with ability to track history of record changes

Availability

Basic Availability Groups
With SQL 2016 Standard Edition

Enhanced AlwaysOn
Distributed availability groups, automatic replica seeding, distributed transactions, automatic failover, load balancing, manageability

Backup enhancements
Managed backup to Azure, Database Recovery Advisor

Scalability

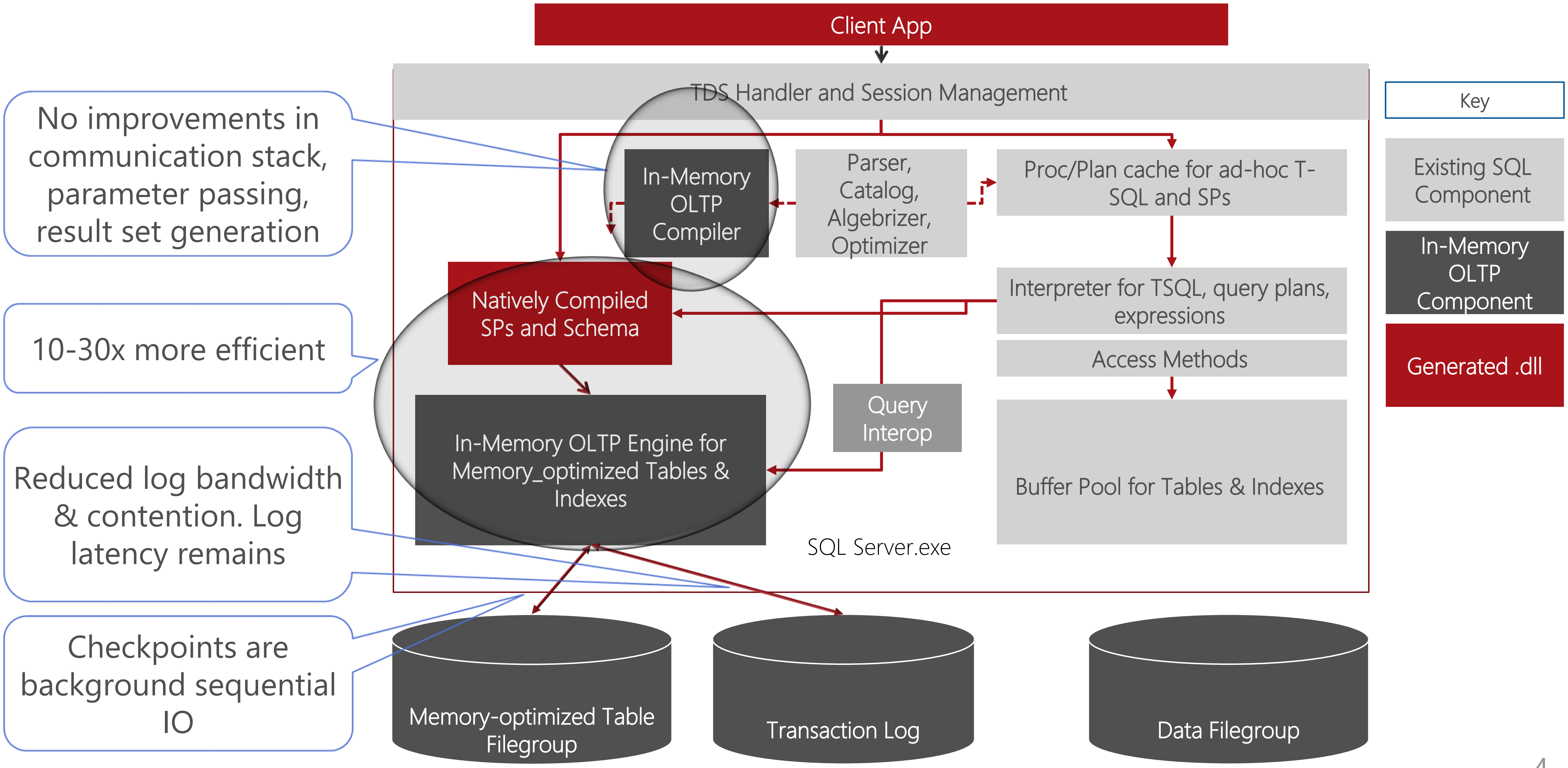
Windows Server support
Support for Windows Server Core and Windows Server ReFS

Live migration
Faster live migration, live migration for non-clustered VMs

Scalability enhancements
Hardware acceleration for TDE, parallelized decryption, TempDB optimization, and more

In-Memory OLTP enhancements

In-memory OLTP Architecture



Performance and Scaling Improvements

Supports up to 2 TB of user data in durable memory optimized tables in a single database.

Multiple threads to persist memory-optimized tables

Parallel Support

- Parallel scan for memory-optimized tables and HASH indexes
- Parallel plan support for accessing memory-optimized tables

Query Surface Area in Native Modules

Disjunction (OR, NOT)

UNION and UNION ALL

SELECT DISTINCT

OUTER JOIN

Subqueries in SELECT statements (EXISTS, IN, scalar subqueries)

Nested execution (EXECUTE) of natively compiled modules

LOB types for parameters and variables.

Natively compiled inline table-valued functions (TVFs)

EXECUTE AS CALLER support

Built-in security functions

Increased support for built-in math functions

Transact-SQL

Support with memory-optimized tables for:

NULLable index key columns.

LOB types [varchar(max), nvarchar(max), and varbinary(max)]

UNIQUE indexes in memory-optimized tables.

FOREIGN KEY constraints between memory-optimized tables.

CHECK and UNIQUE constraints

Triggers (AFTER) for INSERT/UPDATE/DELETE operations.

Transact-SQL

```
ALTER TABLE Sales.SalesOrderDetail
   ALTER INDEX PK_SalesOrderID
   REBUILD
      WITH (BUCKET_COUNT=100000000)
```

ALTER support

Full schema change support: add/alter/drop column/constraint

Add/drop index supported

The `ALTER TABLE` syntax is used for making changes to the table schema, as well as for adding, deleting, and rebuilding indexes

Indexes are considered part of the table definition

Key advantage is the ability to change the `BUCKET_COUNT` with an `ALTER INDEX` statement

Altering natively compiled stored procedures

```
CREATE PROCEDURE [dbo].[usp_1]
WITH NATIVE_COMPILATION, SCHEMABINDING, EXECUTE AS OWNER
AS BEGIN ATOMIC WITH
(
 TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE =
N'us_english'
)
 SELECT c1, c2 from dbo.T1
END
GO

ALTER PROCEDURE [dbo].[usp_1]
WITH NATIVE_COMPILATION, SCHEMABINDING, EXECUTE AS OWNER
AS BEGIN ATOMIC WITH
(
 TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE =
N'us_english'
)
 SELECT c1 from dbo.T1
END
GO
```

You can now perform `ALTER` operations on natively compiled stored procedures using the `ALTER PROCEDURE` statement

Use `sp_recompile` to recompile stored procedures on the next execution

Transact-SQL

Full support for all Collation and Unicode Support

(var)char columns can use any code page supported by SQL Server

Character columns in index keys can use any SQL Server collation

Expressions in natively compiled modules as well as constraints on memory-optimized tables can use any SQL Server collation

Scalar User-Defined Functions for In-Memory OLTP

Create, drop, and alter natively compiled, scalar user-defined functions

Native compilation improves performance of the evaluation of UDFs in T-SQL

Cross-Feature Support

System-Versioned Temporal Tables

Query Store

Row-Level Security (RLS)

Multiple Active Result Sets (MARS)

Transparent Data Encryption (TDE)

Using multiple active result sets (MARS)

MARS simplifies application design :

Applications can have multiple default result sets open and can interleave reading from them.

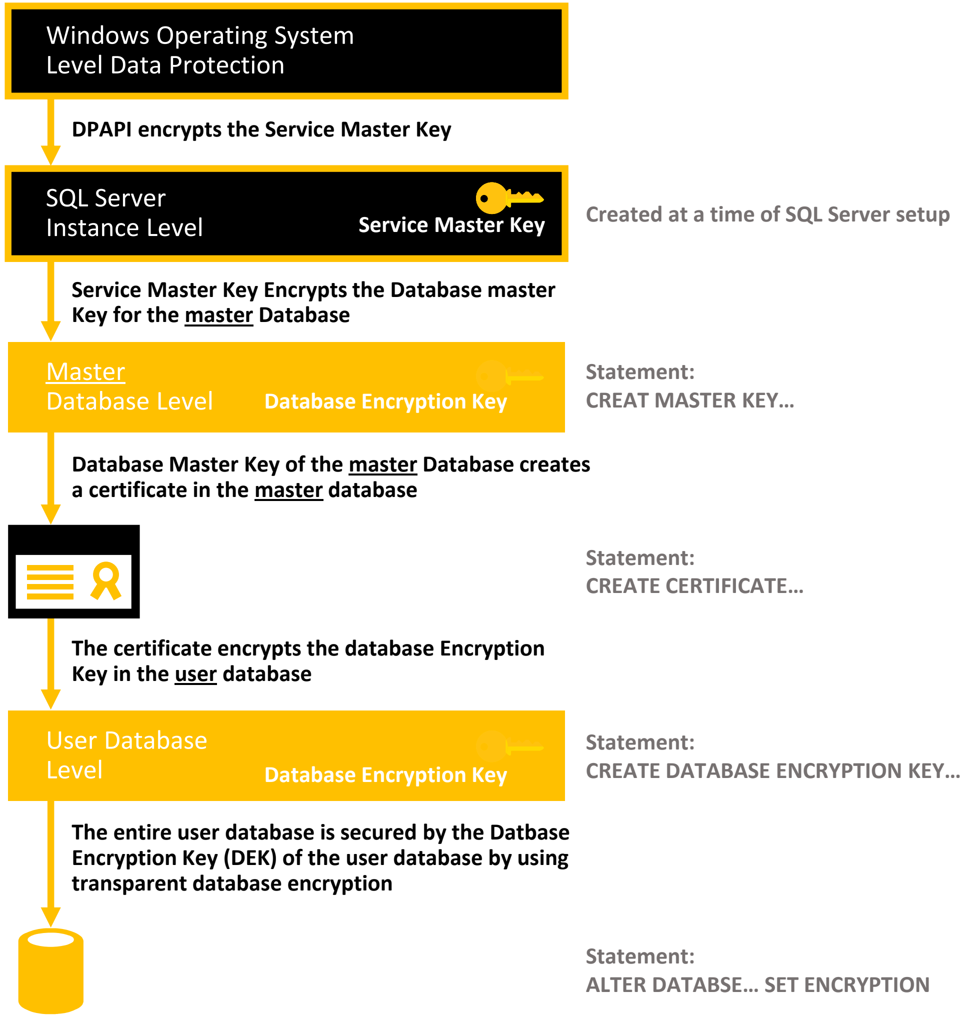
Applications can execute other statements (for example, INSERT, UPDATE, DELETE, and stored procedure calls) while default result sets are open.

Set up MARS connection for memory-optimized tables using `MultipleActiveResultsSets=True` in your connection string

```
Data Source=MSSQL; Initial Catalog=AdventureWorks;
Integrated Security=SSPI;
MultipleActiveResultSets=True
```

Support for Transparent Data Encryption (TDE)

Transparent Database Encryption architecture



In SQL Server 2016, the storage for memory-optimized tables will be encrypted as part of enabling TDE on the database

Simply follow the same steps as you would for a disk-based database

Improvements in Management Studio

Lightweight performance analysis

Transaction Performance Analysis report pinpoints hotspots in the application

Generating migration checklists

Migration checklists show unsupported features used in current disk-based tables and interpreted T-SQL stored procedures

Generated checklists for all or some tables and procedures

Use GUI or PowerShell

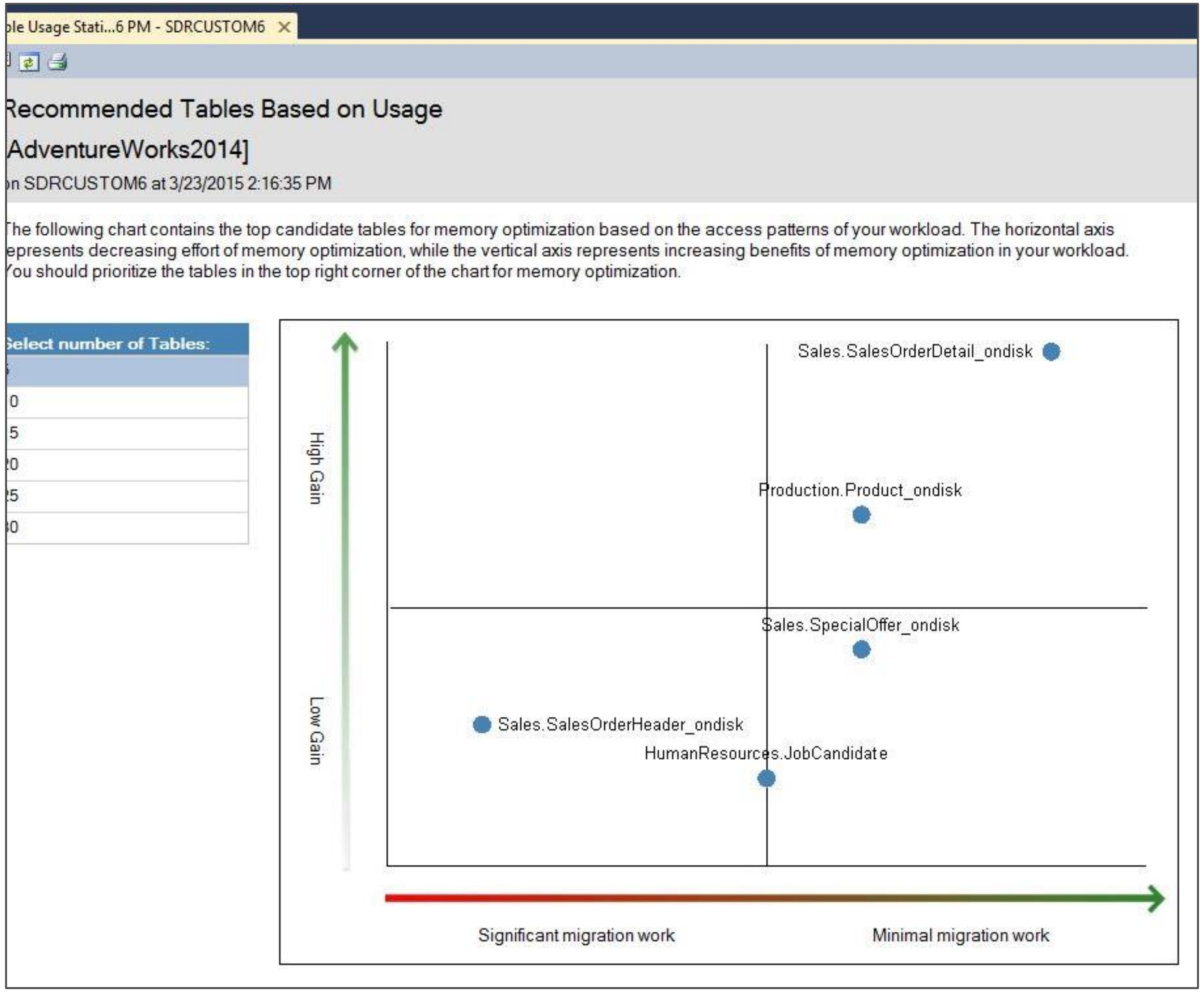
Improved scaling

Other enhancements include:

- Multiple threads to persist memory-optimized tables
- Multi-threaded recovery
- MERGE operation
- Dynamic management view improvements to `sys.dm_db_xtp_checkpoint_stats` and `sys.dm_db_xtp_checkpoint_files`
- `DBCC CHECKDB performance changed to support 1 TB database check improvement by` **7x**

In-Memory OLTP engine has been enhanced to scale linearly on servers up to 4 sockets

New Transaction Performance Analysis Overview report



New report replaces the need to use the Management Data Warehouse to analyze which tables and stored procedures are candidates for in-memory optimization

Summary: In-Memory OLTP enhancements

Capability

ALTER support for memory-optimized tables

Greater Transact-SQL coverage

Benefits

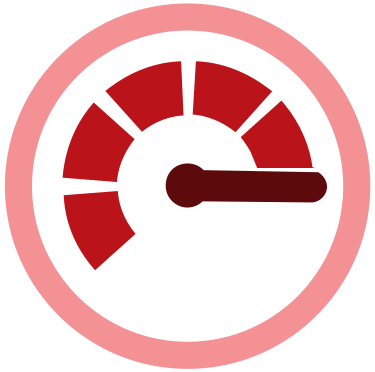
Improved scaling: In-Memory OLTP engine has been enhanced to scale linearly on servers up to 4 sockets

Tooling improvements in Management Studio

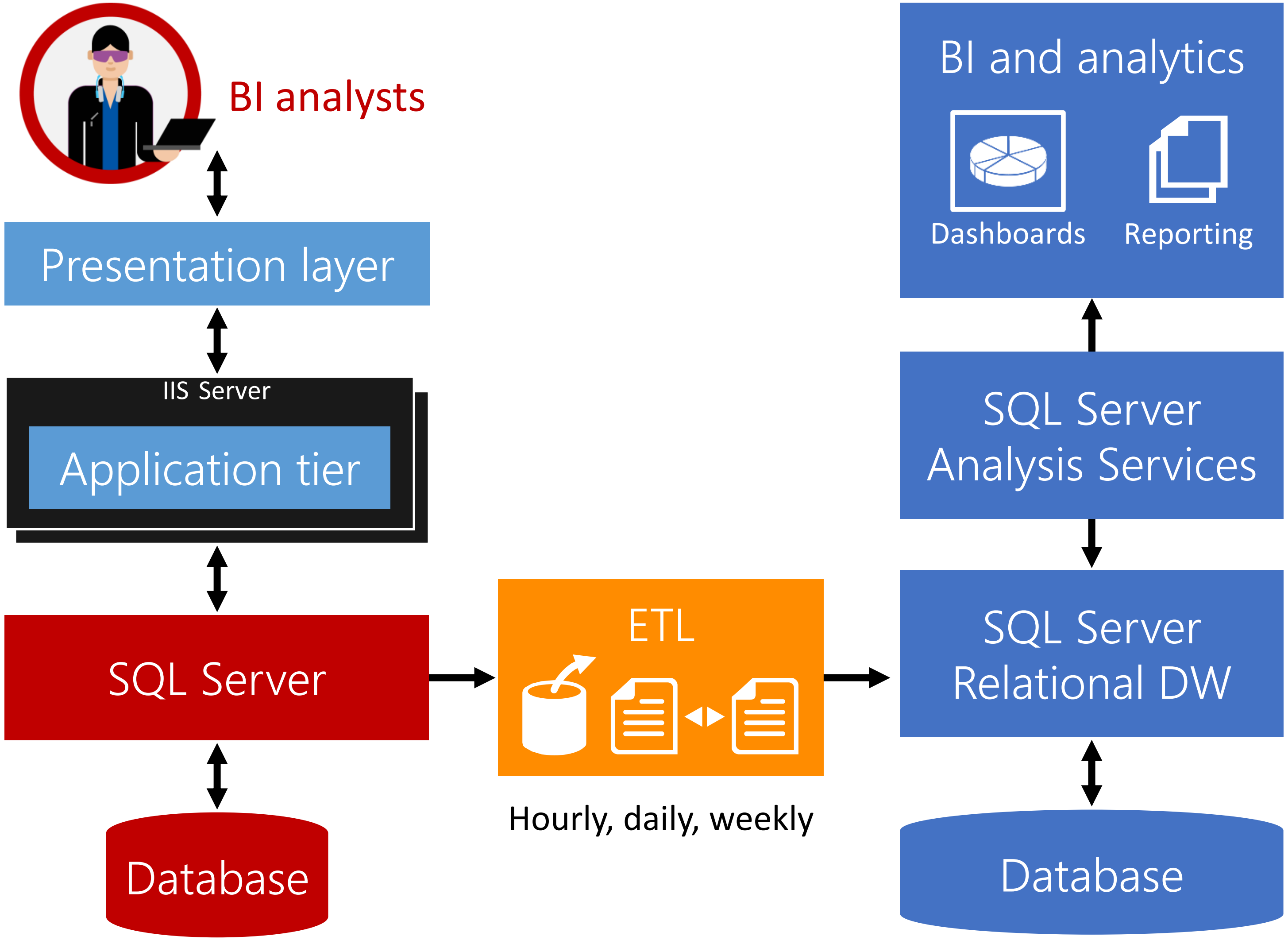
MARS (multiple active result sets) support

TDE (Transparent Data Encryption)-enabled: all on-disk data files are now encrypted once TDE is enabled

Operational Analytics: disk-based and in-memory tables



Traditional operational/analytics architecture



Key issues

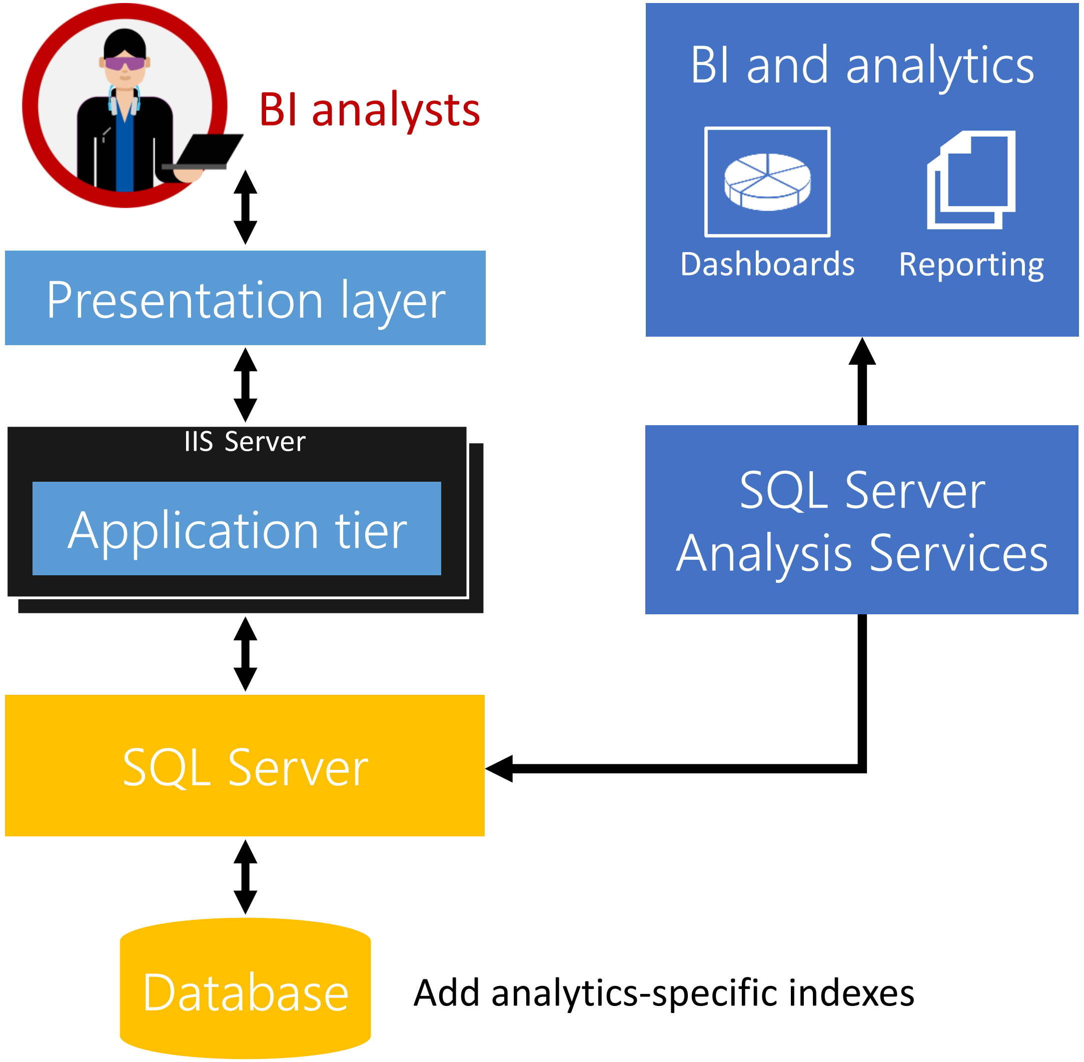
Complex implementation

Requires two servers (capital expenditures and operational expenditures)

Data latency in analytics

High demand;
requires real-time analytics

Minimizing data latency for analytics



Add analytics-specific indexes

Challenges

Analytics queries are resource intensive and can cause blocking

Minimizing impact on operational workloads

Sub-optimal execution of analytics on relational schema

Benefits

No data latency

No ETL

No separate data warehouse

Operational Analytics

The ability to run analytics queries concurrently with operational workloads using the same schema

Goals:

- Minimal impact on operational workloads with concurrent analytics
- Performance analytics for operational schema

Not a replacement for:

- Extreme analytics performance queries possible only using customized schemas (e.g. Star/Snowflake) and pre-aggregated cubes
- Data coming from non-relational sources
- Data coming from multiple relational sources requiring integrated analytics

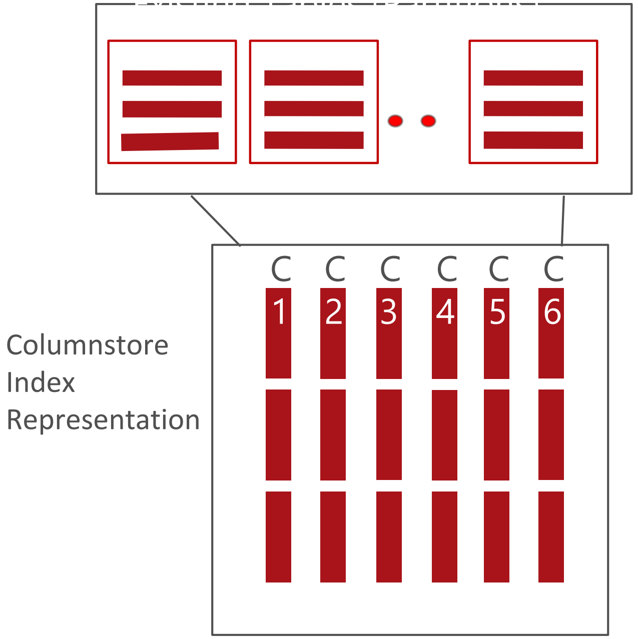
Operational Analytics: disk-based tables

In-Memory In the Data Warehouse
Data Stored Row-Wise: Heaps, b-Trees, Key-Value

- In-Memory ColumnStore
- Both memory and disk
- Built-in to core RDBMS engine
- Customer Benefits:
    - 10-100x faster
    - Reduced design effort
    - Work on customers' existing hardware
    - Easy upgrade; Easy deployment

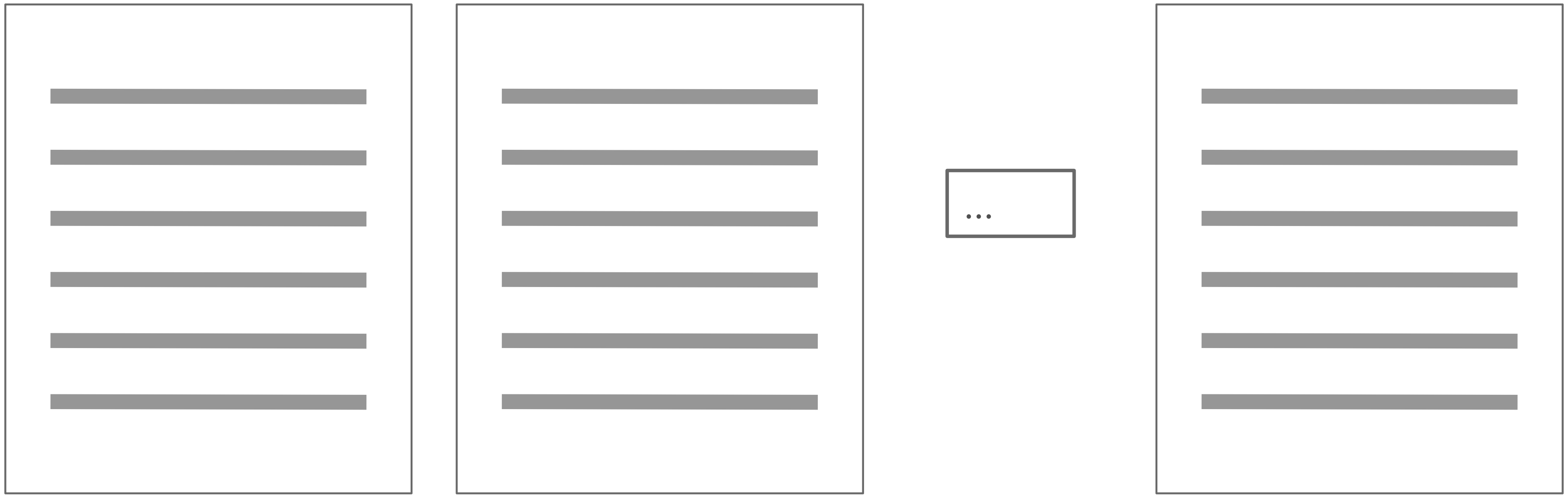
"By using SQL Server 2012 In-Memory ColumnStore, we were able to extract about 100 million records in **2 or 3 seconds** versus the **30 minutes required** previously. "

- Atsuo Nakajima Asst Director, Bank of Nagoya



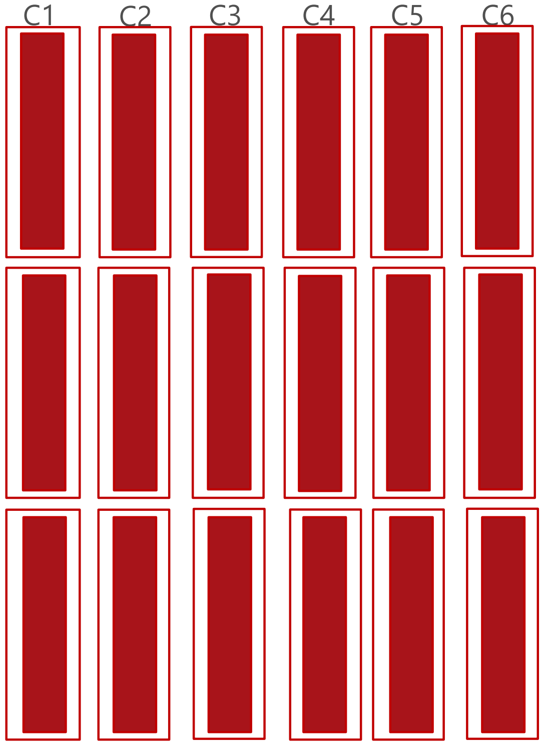
Traditional Storage Models
Data Stored Row-Wise: Heaps, b-Trees, Key-Value

- Relational, dimensional, map reduce



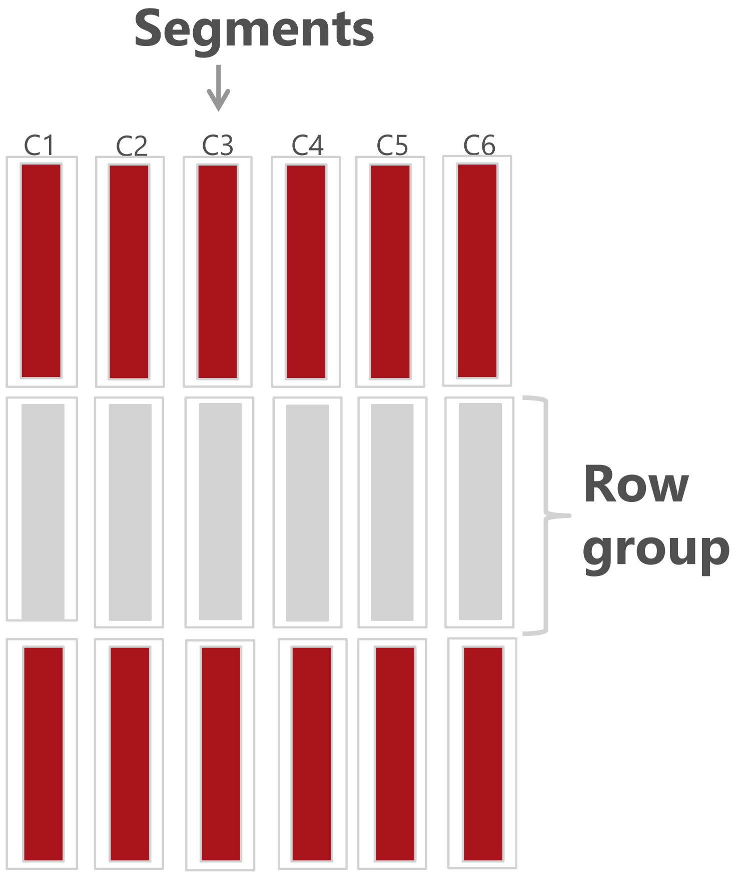
In-Memory DW Storage Model
Data Stored Column-wise

- Each page stores data from a single column
- Highly compressed
  - More data fits in memory
- Each column can be accessed independently
  - Fetch only columns needed
  - Can dramatically decrease I/O



In-Memory DW Index Structure
Row Groups & Segments

- A segment contains values for one column for a set of rows
- Segments for the same set of rows comprise a row group
- Segments are compressed
- Each segment stored in a separate LOB
- Segment is unit of transfer between disk and memory



In-Memory DW Index Processing an Example

OrderDateKey	ProductKey	StoreKey	RegionKey	Quantity	SalesAmount
20101107	106	01	1	6	30.00
20101107	103	04	2	1	17.00
20101107	109	04	2	2	20.00
20101107	103	03	2	1	17.00
20101107	106	05	3	4	20.00
20101108	106	02	1	5	25.00
20101108	102	02	1	1	14.00
20101108	106	03	2	5	25.00
20101108	109	01	1	1	10.00
20101109	106	04	2	4	20.00
20101109	106	04	2	5	25.00
20101109	103	01	1	1	17.00

Horizontally Partition Row Groups

OrderDateKey	ProductKey	StoreKey	RegionKey	Quantity	SalesAmount
20101107	106	01	1	6	30.00
20101107	103	04	2	1	17.00
20101107	109	04	2	2	20.00
20101107	103	03	2	1	17.00
20101107	106	05	3	4	20.00
20101108	106	02	1	5	25.00

OrderDateKey	ProductKey	StoreKey	RegionKey	Quantity	SalesAmount
20101108	102	02	1	1	14.00
20101108	106	03	2	5	25.00
20101108	109	01	1	1	10.00
20101109	106	04	2	4	20.00
20101109	106	04	2	5	25.00
20101109	103	01	1	1	17.00

Vertical Partition Segments

OrderDateKey	ProductKey	StoreKey	RegionKey	Quantity	SalesAmount
20101107	106	01	1	6	30.00
20101107	103	04	2	1	17.00
20101107	109	04	2	2	20.00
20101107	103	03	2	1	17.00
20101107	106	05	3	4	20.00
20101108	106	02	1	5	25.00

OrderDateKey	ProductKey	StoreKey	RegionKey	Quantity	SalesAmount
20101108	102	02	1	1	14.00
20101108	106	03	2	5	25.00
20101108	109	01	1	1	10.00
20101109	106	04	2	4	20.00
20101109	106	04	2	5	25.00
20101109	103	01	1	1	17.00

Compress Each Segment*
Some Compress More than Others

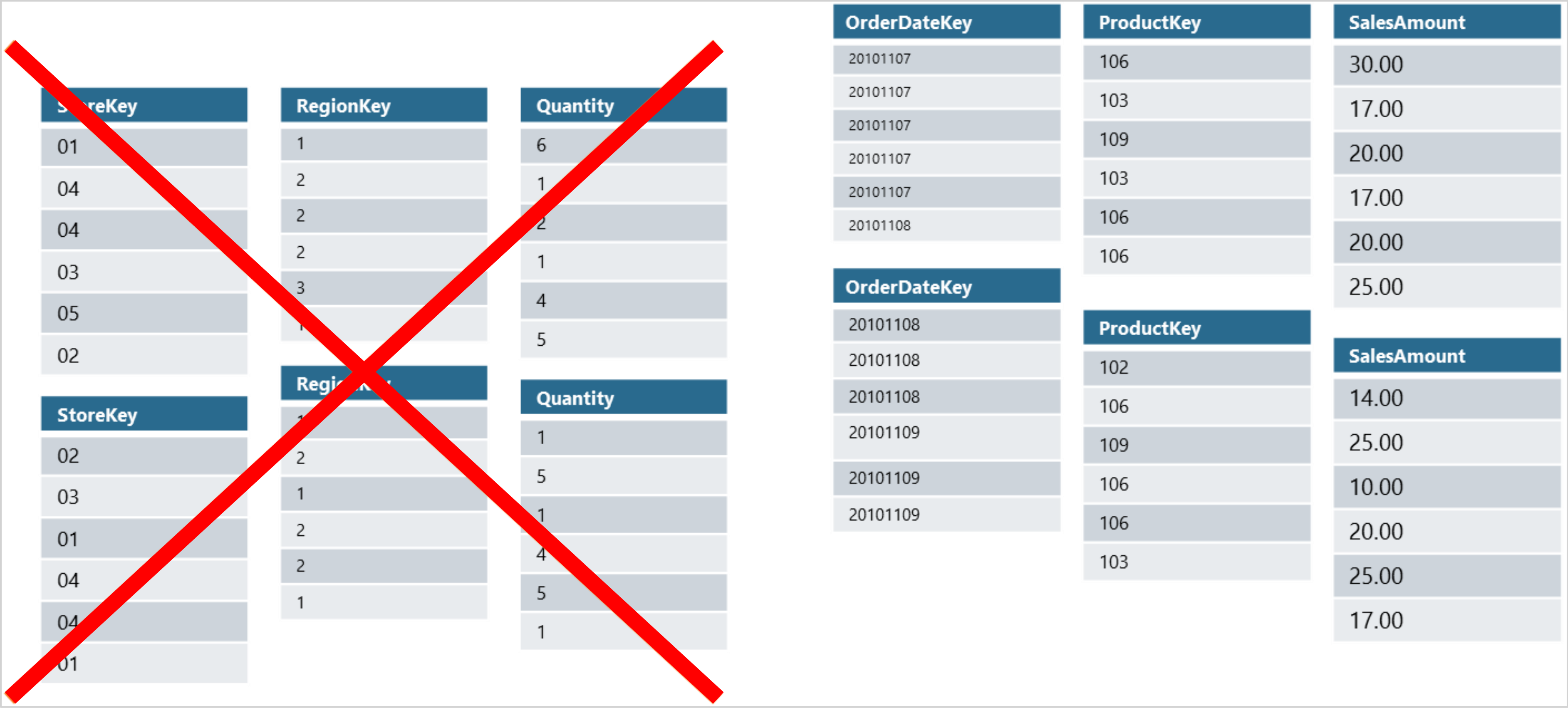
OrderDateKey	ProductKey	StoreKey	RegionKey	Quantity	SalesAmount
20101107	106	01	1	6	30.00
20101107	103	04	2	1	17.00
20101107	109	04	2	2	20.00
20101107	103	03	2	1	17.00
20101107	106	05	3	4	20.00
20101108	106	02	1	5	25.00

OrderDateKey	ProductKey	StoreKey	RegionKey	Quantity	SalesAmount
20101108	102	02	1	1	14.00
20101108	106	03	2	5	25.00
20101108	109	01	1	1	10.00
20101109	106	04	2	4	20.00
20101109	106	04	2	5	25.00
20101109	103	01	1	1	17.00

*Encoding and reordering not shown

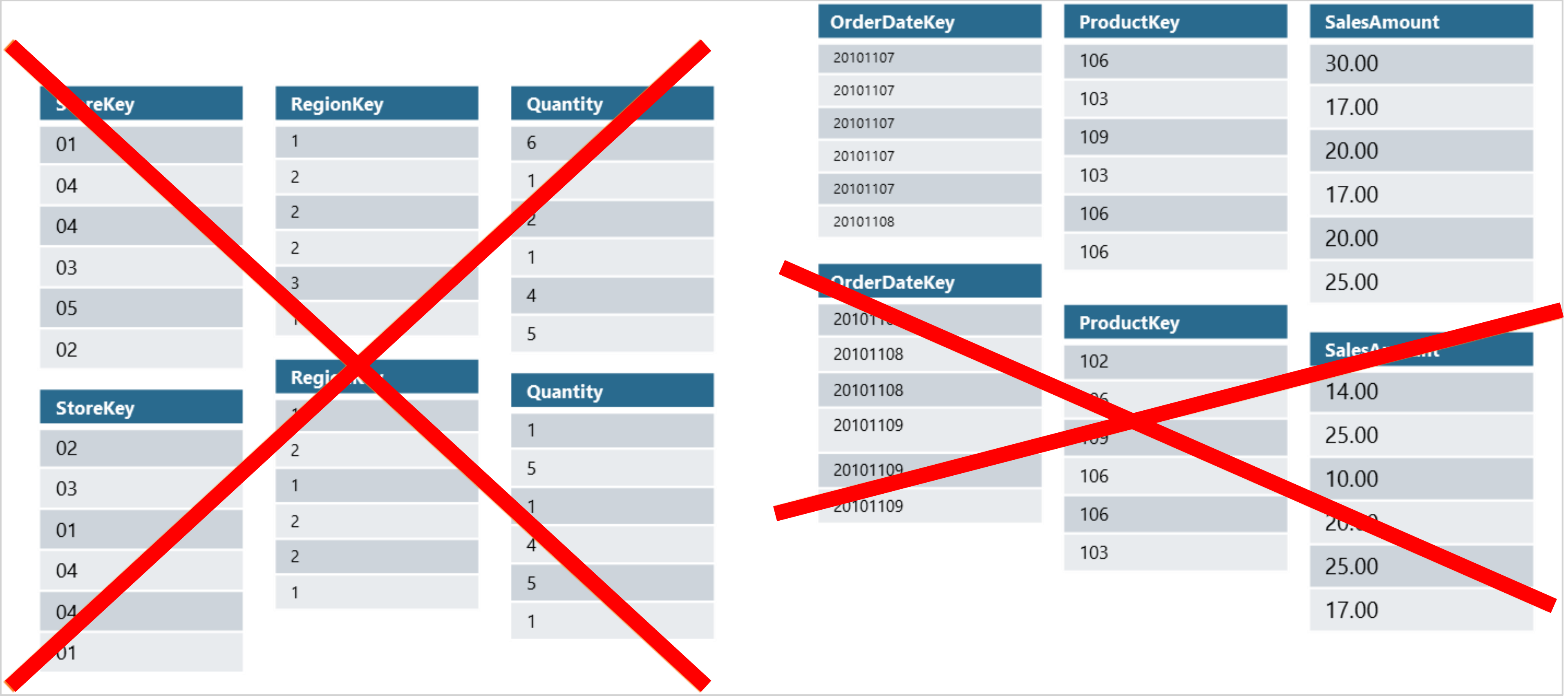
Fetch Only Needed Columns
Segment Elimination

```sql
SELECT ProductKey, SUM (SalesAmount)
FROM SalesTable
WHERE OrderDateKey < 20101108
```

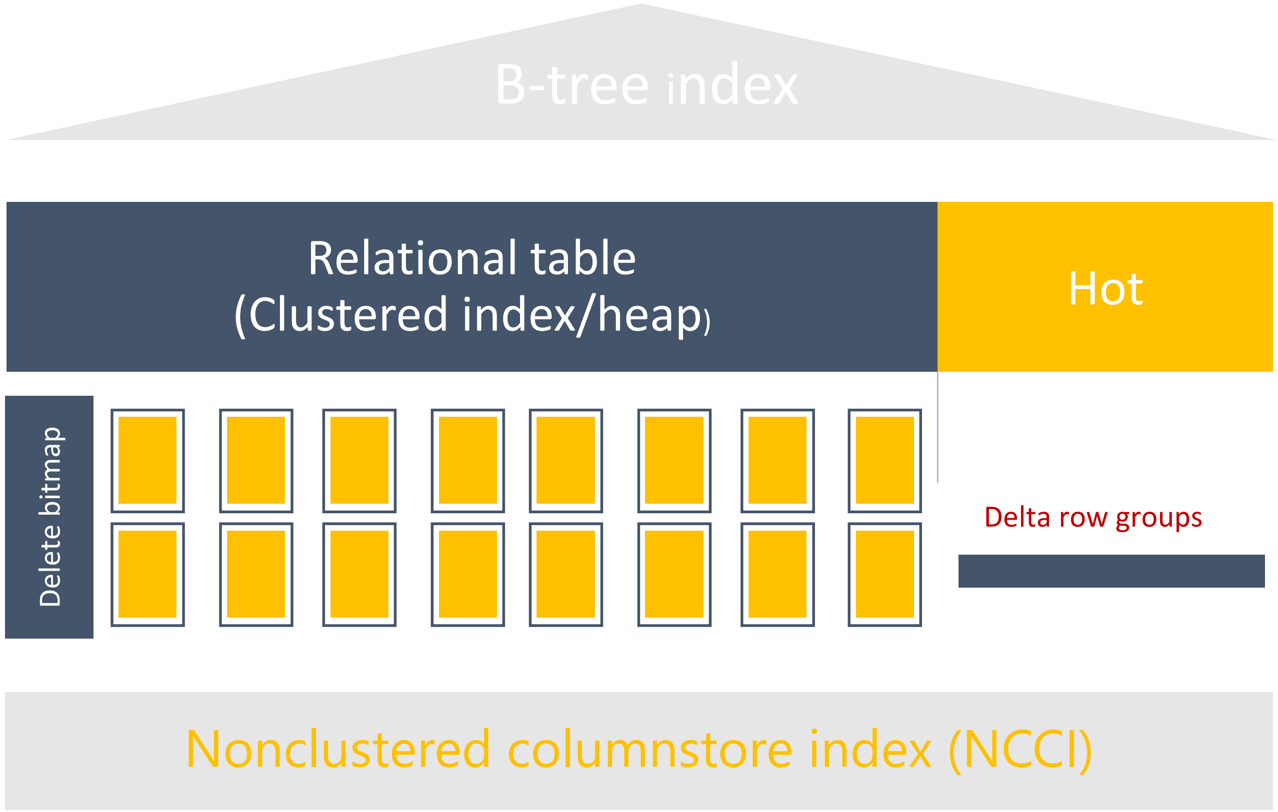


Fetch Only Needed Segments
Segment Elimination

```sql
SELECT ProductKey, SUM (SalesAmount)
FROM SalesTable
WHERE OrderDateKey < 20101108
```



Operational Analytics with columnstore index



Key points

Create an updateable NCCI for analytics queries

Drop all other indexes that were created for analytics

No application changes

Columnstore index is maintained just like any other index

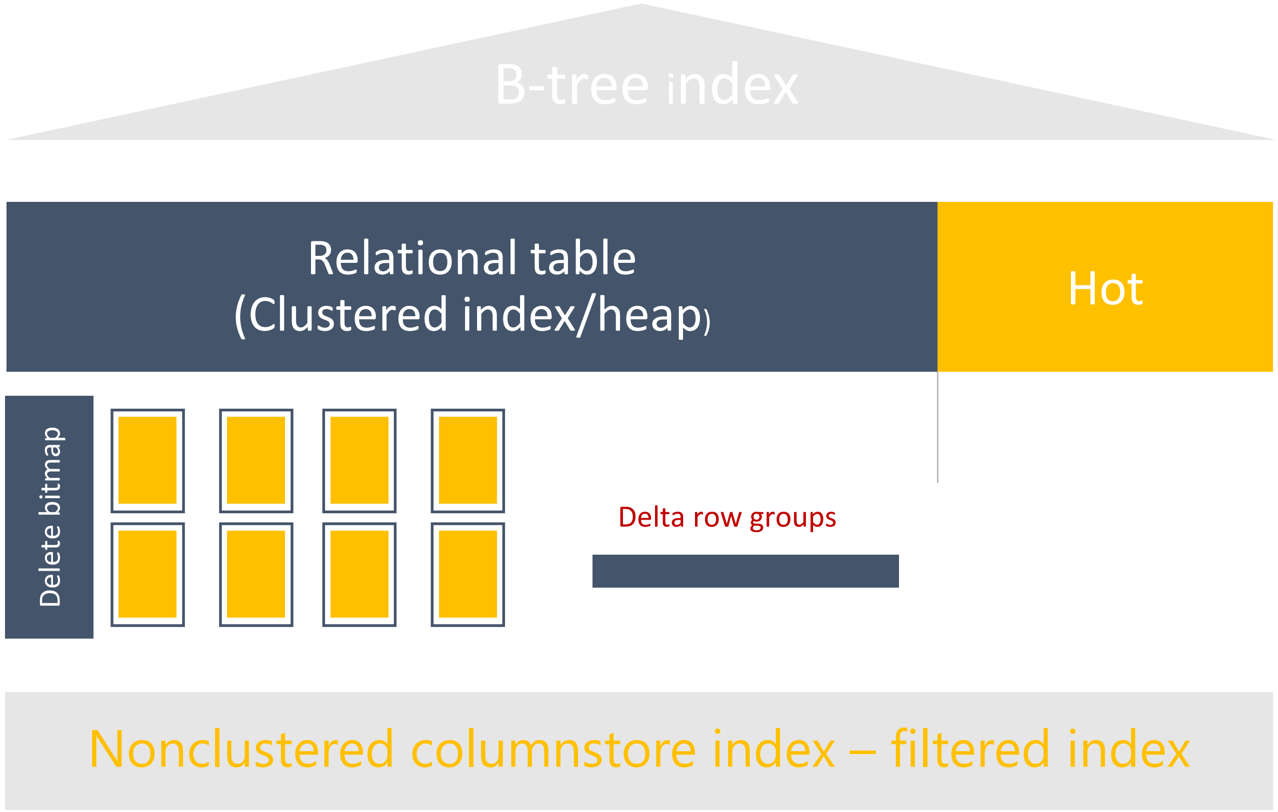
Query optimizer will choose columnstore index where needed

Operational Analytics with columnstore index

DML operations on OLTP workload

Operation	B-tree (NCI)	Non-clustered columnstore index (NCCI)
Insert	Insert row into B-tree.	Insert row into B-tree (delta store).
Delete	(a) Seek row(s) to be deleted. (b) Delete the row.	(a) Seek row in delta stores. (There can be multiple rows.) (b) If found, delete row. (c) If not found, insert key into delete row buffer.
Update	(a) Seek the row(s). (b) Update.	(a) Delete row (steps same as above). (b) Insert updated row into delta store.

Operational Analytics: minimizing columnstore overhead



Key points

Create columnstore only on cold data by using filtered predicate to minimize maintenance

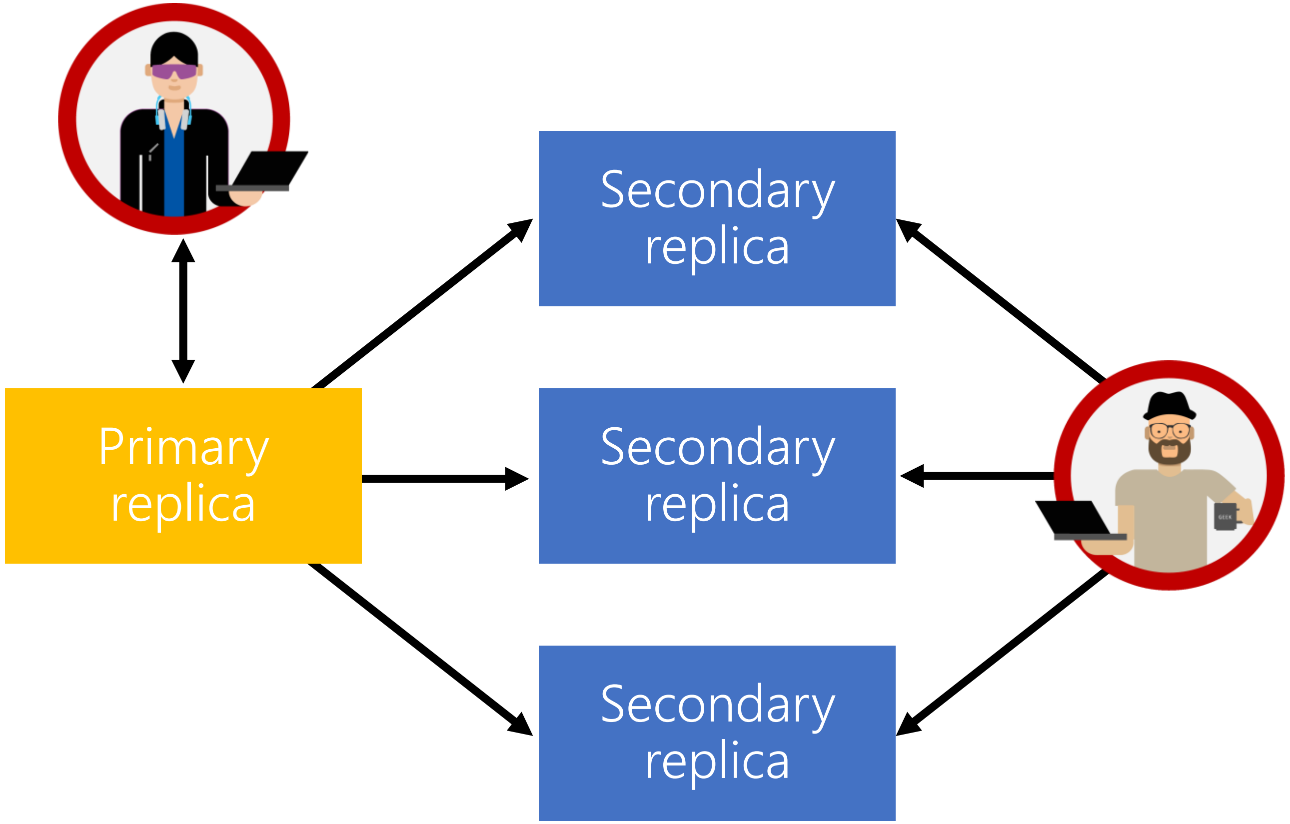
Analytics query accesses both columnstore and ‘hot’ data transparently

Example:

Order Management Application: create nonclustered columnstore index where order_status = ‘SHIPPED’

Using Availability Groups instead of data warehouses

Always On Availability Group



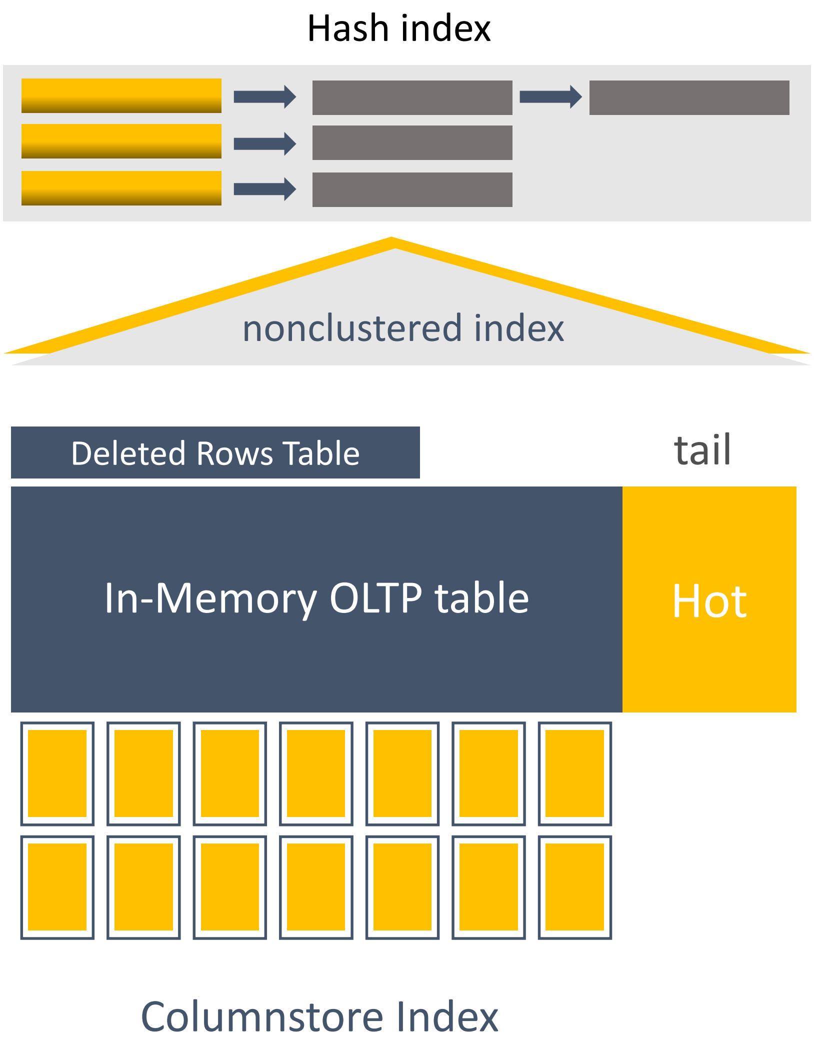
Key points

Mission-critical operational workloads typically configured for high availability using Always On Availability Groups

You can offload analytics to readable secondary replica

Operational Analytics: In-Memory Tables

Operational Analytics: columnstore on In-Memory Tables



No explicit delta row group

Rows (tail) not in columnstore stay in In-Memory OLTP table

No columnstore index overhead when operating on tail

Background task migrates rows from tail to columnstore in chunks of 1 million rows

Deleted Rows Table (DRT) – Tracks deleted rows

Columnstore data fully resident in memory

Persisted together with operational data

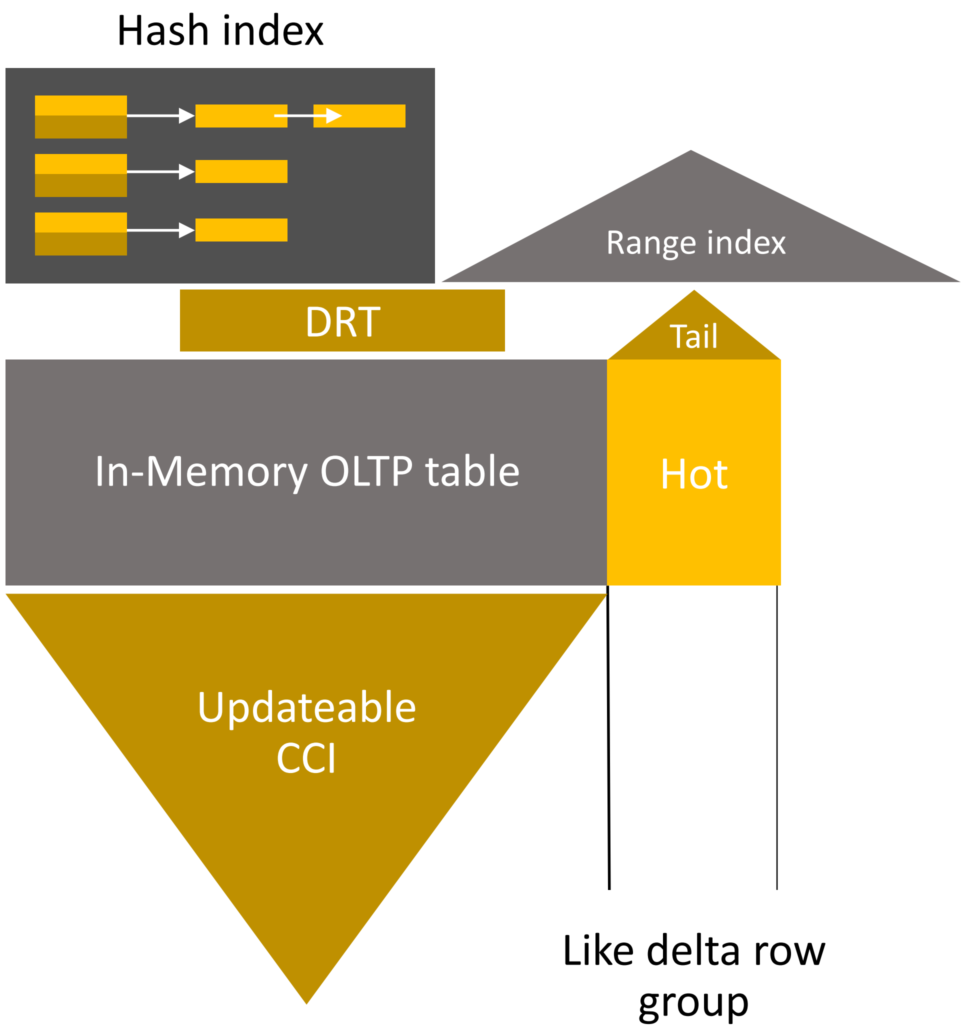
No application changes required

Operational Analytics: columnstore overhead

DML operations on In-Memory OLTP

Operation	Hash or range index	HK-CCI
Insert	Insert row into HK.	Insert row into HK.
Delete	(a) Seek row(s) to be deleted. (b) Delete the row.	(a) Seek row(s) to be deleted. (b) Delete the row in HK. (c) If row in TAIL, then return. If not, insert <colstore-RID> into DRT.
Update	(a) Seek the row(s) to be updated. (b) Update (delete/insert).	(a) Seek the row(s) to be updated. (b) Update (delete/insert) in HK. (c) If row in TAIL, then return If not, insert <colstore-RID> into DRT.

Operational Analytics: minimizing columnstore overhead



DML operations

Keep hot data only in in-memory tables
Example: data stays hot for 1 day, 1 week…

Workaround:
Use TF – 9975 to disable auto-compression
Force compression using a spec-proc
“sp_memory_optimized_cs_migration”

Analytics queries

Offload analytics to AlwaysOn readable secondary

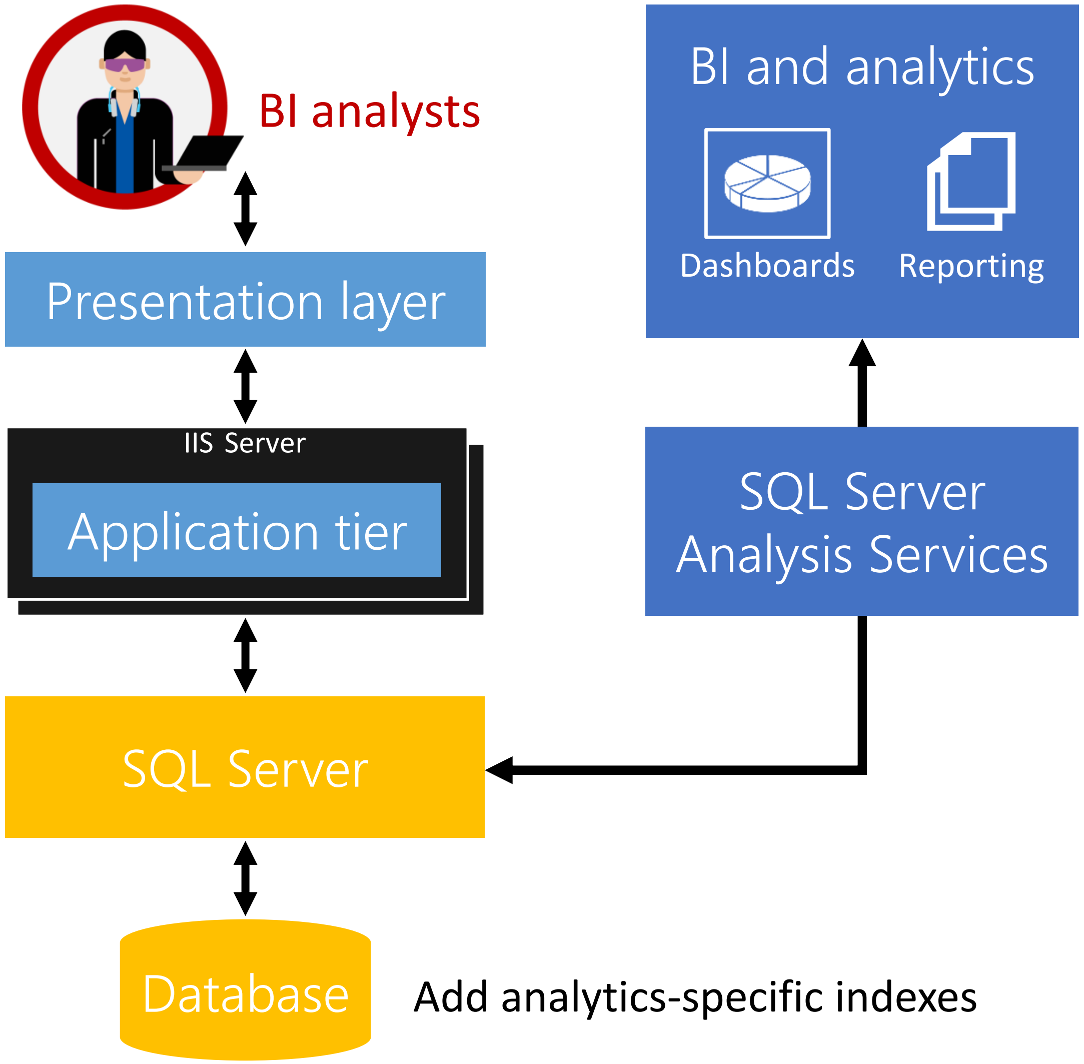
Summary of improvements

Improvements	SQL Server 2014	SQL Server 2016
clustered columnstore index	Master copy of the data (10x compression) Only index supported; simplified analytics No PK/FK constraints Uniqueness can be enforced through materialized views Locking granularity for UPDATE/DELETE at row group level DDL: ALTER, REBUILD, REORGANIZE	Master copy of the data (10x compression) Additional B-tree indexes for efficient equality, short-range searches, and PK/FK constraints Locking granularity at row level using NCI index path DDL: ALTER, REBUILD, REORGANIZE
updateable non-clustered index	Introduced in SQL Server 2012 NCCI is **read-only**: no delete bitmap or delta store Optimizer will choose between NCCI and NCI(s)/CI or heap-based on the cost-based model Partitioning supported	Updateable Ability to mix OLTP and analytics workload Ability to create filtered NCCI Partitioning supported
equality and short-range queries	Row group elimination (when possible) Partition-level scan (somewhat expensive) Full index scan (expensive)	Optimizer can choose NCI on column C1; index points directly to row group No full index scan Covering NCI index
string predicate pushdown	Retrieve 10 million rows by converting dictionary encoded value to string Apply string predicate on 10 million rows	Apply filter on dictionary entries Find rows that refer to dictionary entries that qualify (R1) Find rows not eligible for this optimization (R2) Scan returns (R1 + R2) rows Filter node applies string predicate on (R2) Row returned by Filter node = (R1 + R2’)

Support for index maintenance

Operation	SQL Server 2014	SQL Server 2016
Removing deleted rows	Requires index REBUILD	Index REORGANIZE Remove deleted rows from single compressed RG Merge one or more compressed RGs with deleted rows Done ONLINE
Smaller RG size resulting from: Smaller BATCHSIZE Memory pressure Index build residual	Index REBUILD	Index REORGANIZE
Ordering rows	Create clustered index Create columnstore index by dropping clustered index	No changes
Query	Row group granularity No support for RCSI or SI Recommendation: use read uncommitted	Support of SI and RCSI (non-blocking)
Insert	Lock at row level (trickle insert) Row group level for set of rows	No changes
Delete	Lock at row group level	Row-level lock in conjunction with NCI
Update	Lock at row group level Implemented as Delete/Insert	Row-level lock in conjunction with NCI
AlwaysOn Failover Clustering (FCI)	Fully supported	Fully supported
AlwaysON Availability Groups	Fully supported except readable secondary	Fully supported with readable secondary
Index create/rebuild	Offline	Offline

Summary: Operational Analytics



Add analytics-specific indexes

Capability

Ability to run analytics queries concurrently with operational workloads using the same schema

Data Warehouse queries can be run on In-Memory OLTP workloads with no application changes

Benefits

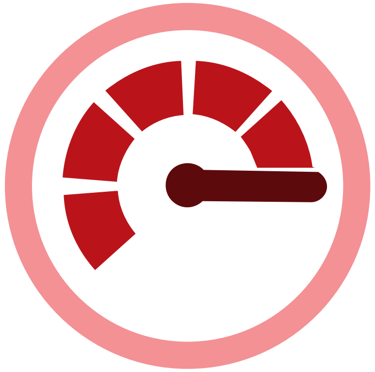
Minimal impact on OLTP workloads

Best performance and scalability available

Offloading analytics workload to readable secondary

Query Store

Your flight data recorder
for your database



Problems with query performance

Website Is down

Database is not working

Temporary perf issues

Impossible to predict / root cause

DB upgraded

Regression caused by new bits

Fixing query plan choice regressions is difficult

- Query plan cache is not well-suited for performance troubleshooting

Long time to detect the issue (TTD)

- Which query is slow? Why is it slow?
- What was the previous plan?

Long time to mitigate (TTM)

- Can I modify the query?
- How to use plan guide?

Plan choice change can cause these problems

The solution: Query Store

Dedicated store for query workload performance data

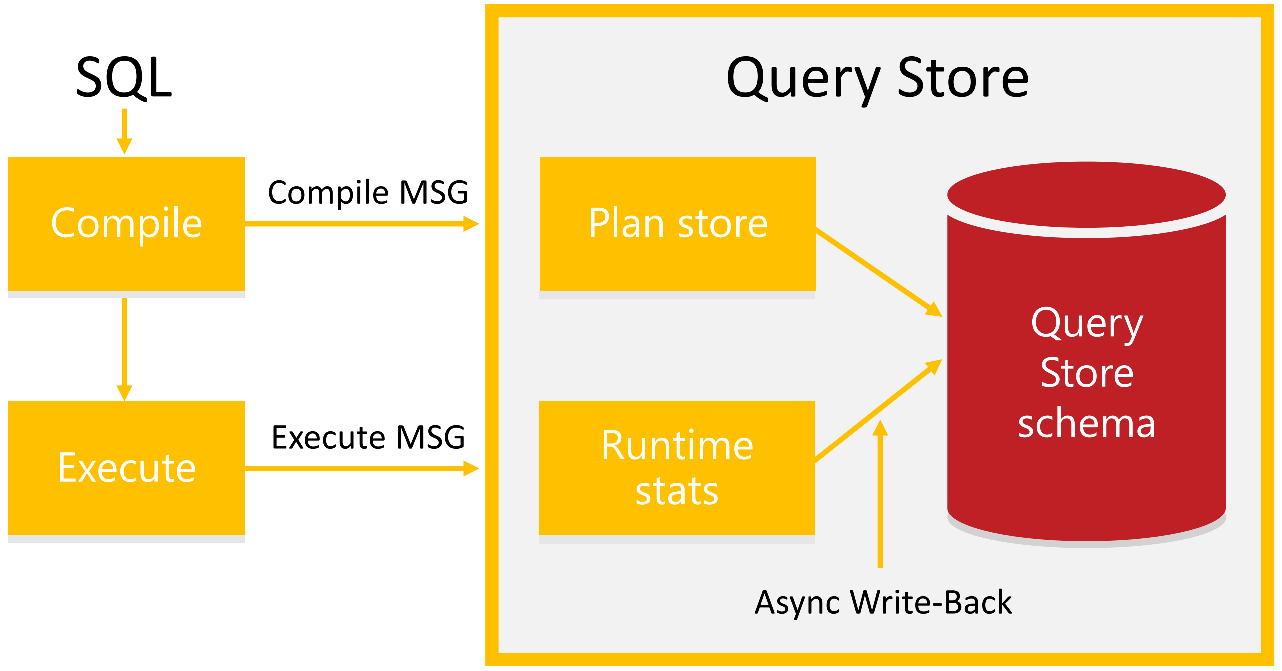
- Captures the history of plans for each query
- Captures the performance of each plan over time
- Persists the data to disk (works across restarts, upgrades, and recompiles)

Significantly reduces TTD/TTM

- Find regressions and other issues in seconds
- Allows you to force previous plans from history

DBA is now in control

Query Store Architecture



Durability latency controlled by DB option
DATA_FLUSH_INTERNAL_SECONDS

Collects query texts (plus all relevant properties)

Stores all plan choices and performance metrics

Works across restarts / upgrades / recompiles

Dramatically lowers the bar for performance troubleshooting

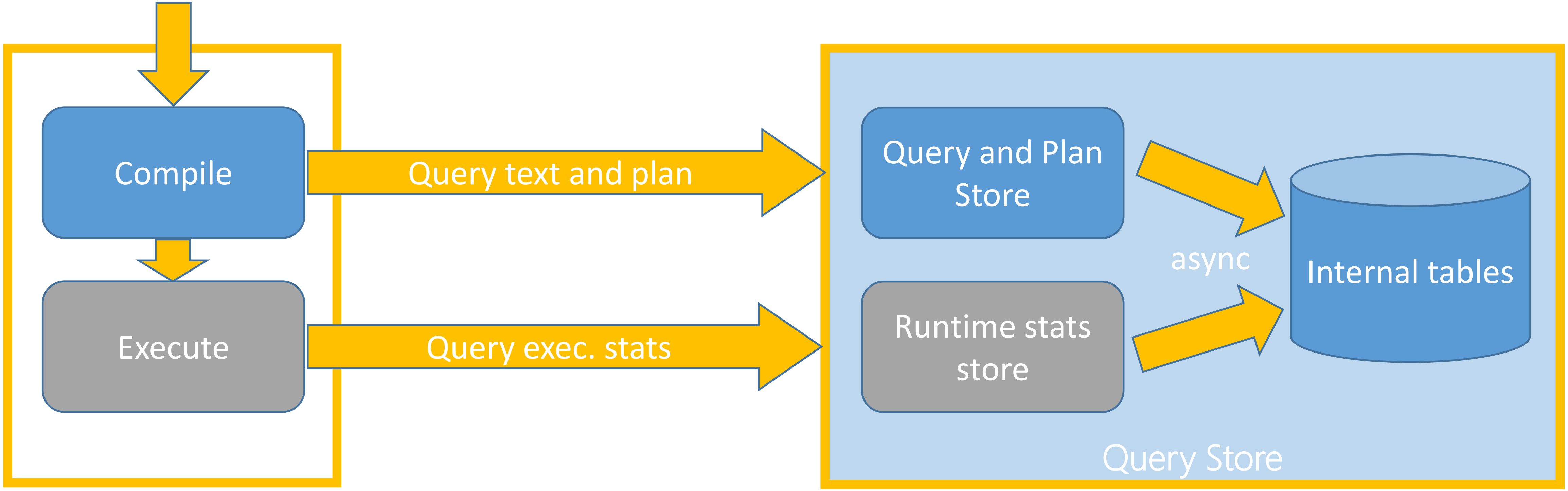
New Views

Intuitive and easy plan forcing

Query Store write architecture

Query Store captures data in-memory to minimize I/O overhead

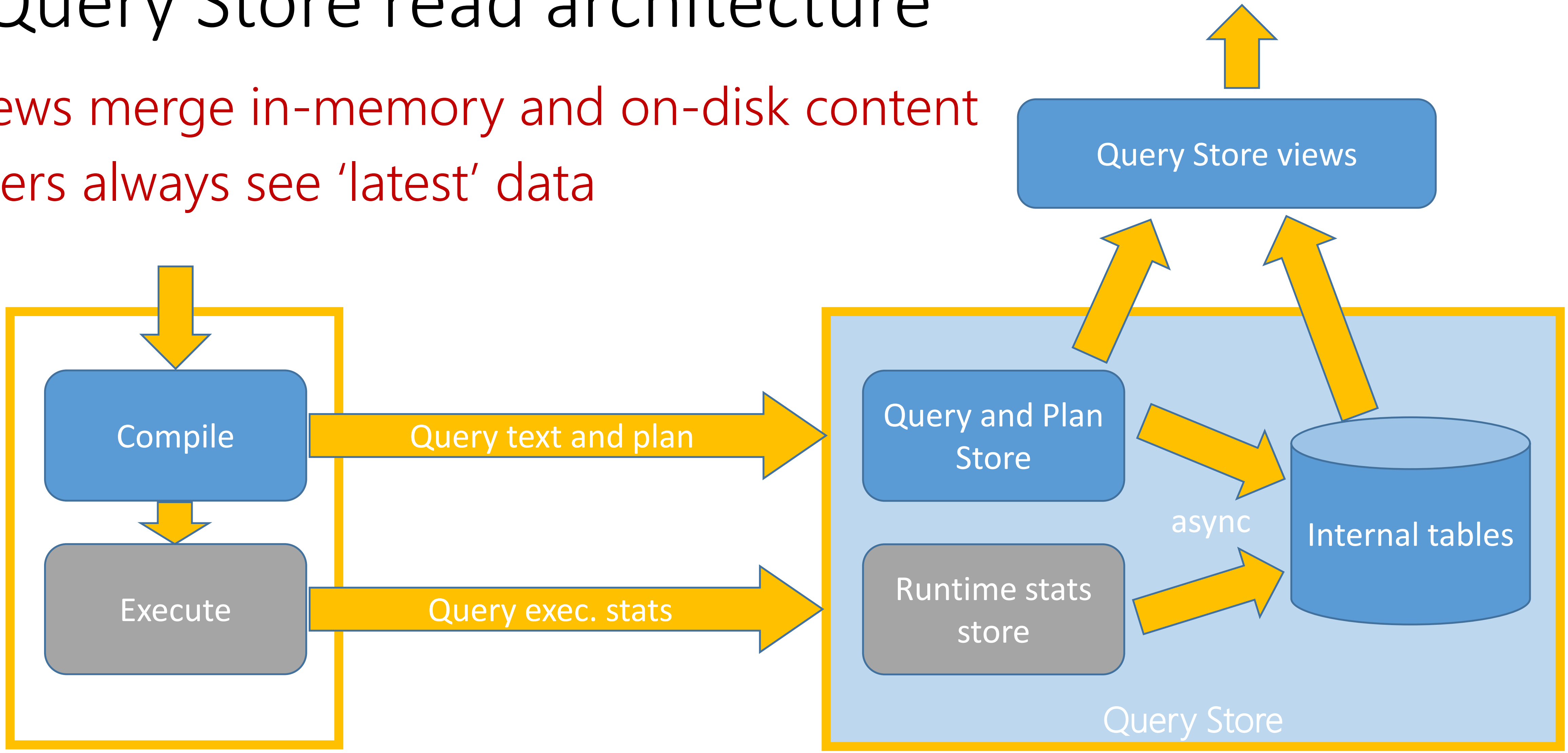
Data is persisted to disk asynchronously in the background



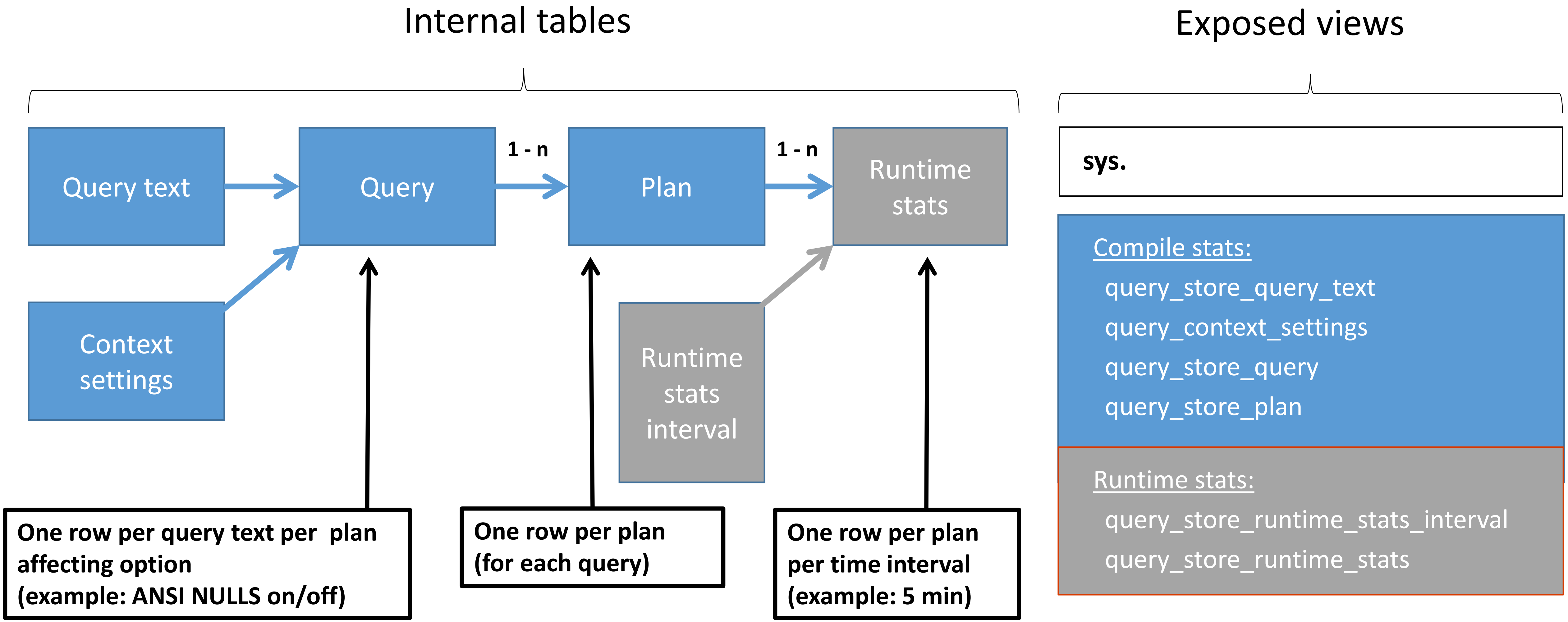
Query Store read architecture

Views merge in-memory and on-disk content

Users always see 'latest' data



Query Store schema explained



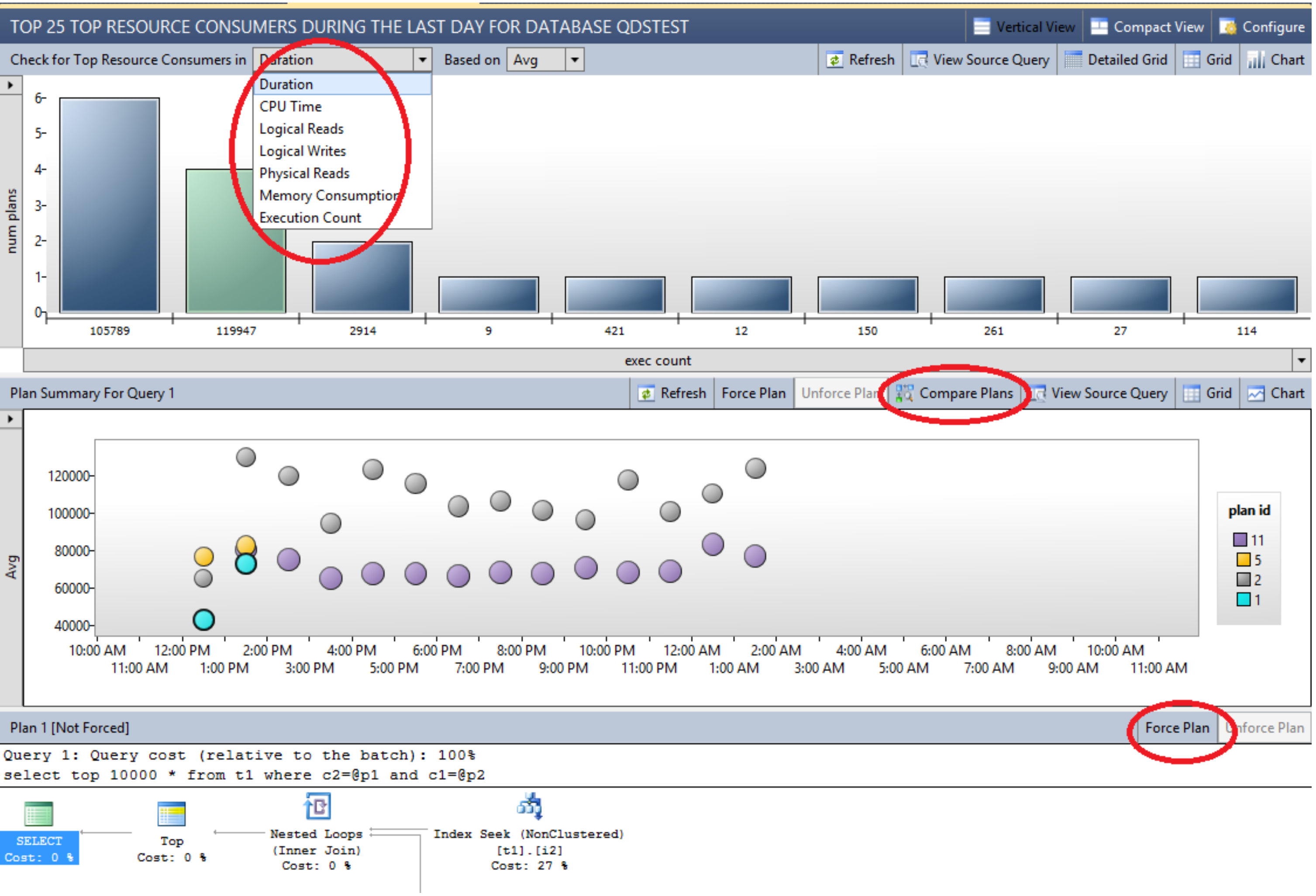
Keeping stability while upgrading to SQL Sever 2016

SQL Server 2016

Query Optimizer (QO) enhancements tied to database compatibility level

Install bits Keep *existing* compat. level	→	Run Query Store (create a baseline)	→	Move to **vNext CompatLevel**	→	Fix regressions with plan forcing ✔

Monitoring performance by using the Query Store



The Query Store feature provides DBAs with insight on query plan choice and performance

Working with Query Store

```
/* (6) Performance analysis using Query Store views*/
SELECT q.query_id, qt.query_text_id, qt.query_sql_text,
SUM(rs.count_executions) AS total_execution_count
FROM
sys.query_store_query_text qt JOIN
sys.query_store_query q ON qt.query_text_id =
q.query_text_id JOIN
sys.query_store_plan p ON q.query_id = p.query_id JOIN
sys.query_store_runtime_stats rs ON p.plan_id = rs.plan_id
GROUP BY q.query_id, qt.query_text_id, qt.query_sql_text
ORDER BY total_execution_count DESC

/* (7) Force plan for a given query */
exec sp_query_store_force_plan
12 /*@query_id*/, 14 /*@plan_id*/
```

```
);

/* (4) Clear all Query Store data */
ALTER DATABASE MyDB SET QUERY_STORE CLEAR;

/* (5) Turn OFF Query Store */
ALTER DATABASE MyDB SET QUERY_STORE = OFF;
```

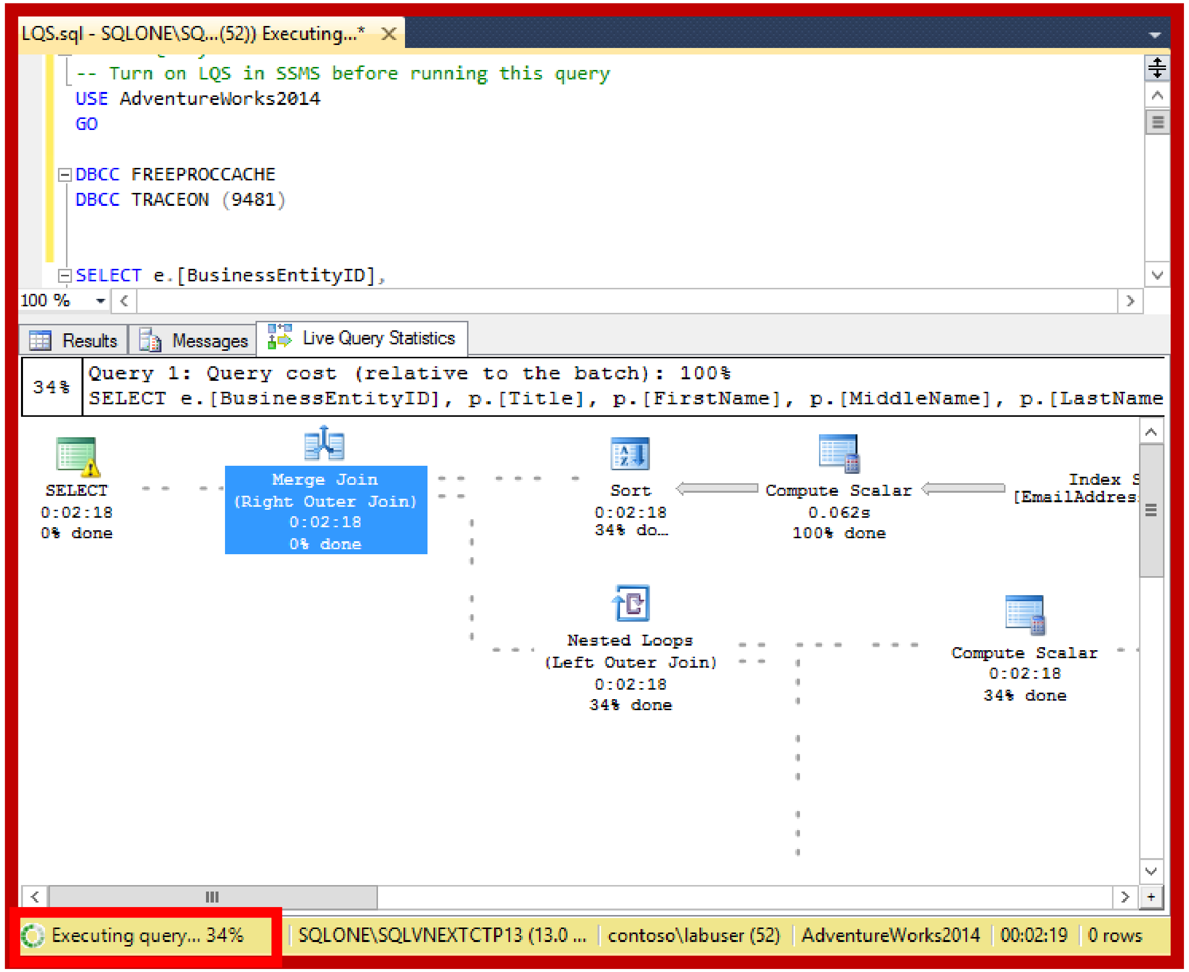
DB-level feature exposed through T-SQL extensions

ALTER DATABASE

Catalog views (settings, compile, and runtime stats)

Stored Procs (plan forcing, query/plan/stats cleanup)

Live query statistics



View CPU/memory usage, execution time, query progress, and more

Enables rapid identification of potential bottlenecks for troubleshooting query performance issues

Allows drill down to live operator level statistics:

Number of generated rows

Elapsed time

Operator progress

Live warnings

Summary: Query Store

Capability

Query Store helps customers quickly find and fix query performance issues

Query Store is a ‘flight data recorder’ for database workloads

Benefits

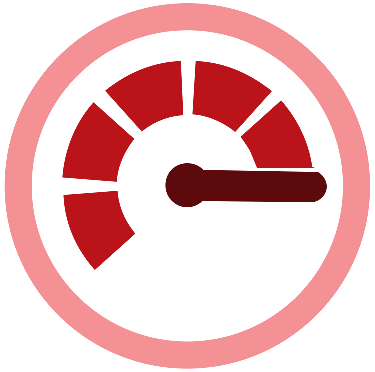
Greatly simplifies query performance troubleshooting

Provides performance stability across SQL Server upgrades

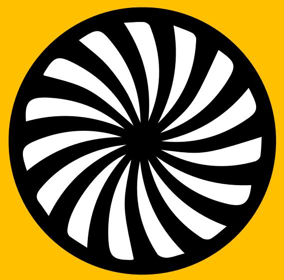
Allows deeper insight into workload performance

Temporal Tables

Query back in time

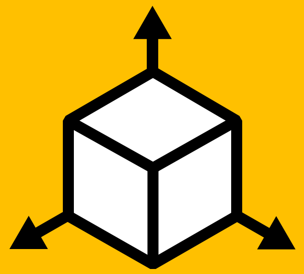


Why temporal



Time travel

Data audit



Slowly changing dimensions

Repair record-level corruptions

Data changes over time

Tracking and analyzing changes is often important

Temporal in DB

Automatically tracks history of data changes

Enables easy querying of historical data states

Advantages over workarounds

Simplifies app development and maintenance

Efficiently handles complex logic in DB engine

How to start with temporal

ANSI 2011
compliant

NO change in programming model

New Insights

CREATE ***temporal*** **TABLE**
PERIOD FOR
SYSTEM_TIME…

ALTER ***regular_table***
TABLE ADD PERIOD…

DDL

DML

INSERT / BULK INSERT

UPDATE

DELETE

MERGE

SELECT * FROM *temporal*

Querying

Temporal Que

FOR SYSTEM_TIME
AS OF
FROM..TO
BETWEEN..AND
CONTAINED IN

Temporal database support: BETWEEN

```
SELECT * FROM
Person.BusinessEntityContact
FOR SYSTEM_TIME BETWEEN @Start AND @End
WHERE ContactTypeID = 17
```

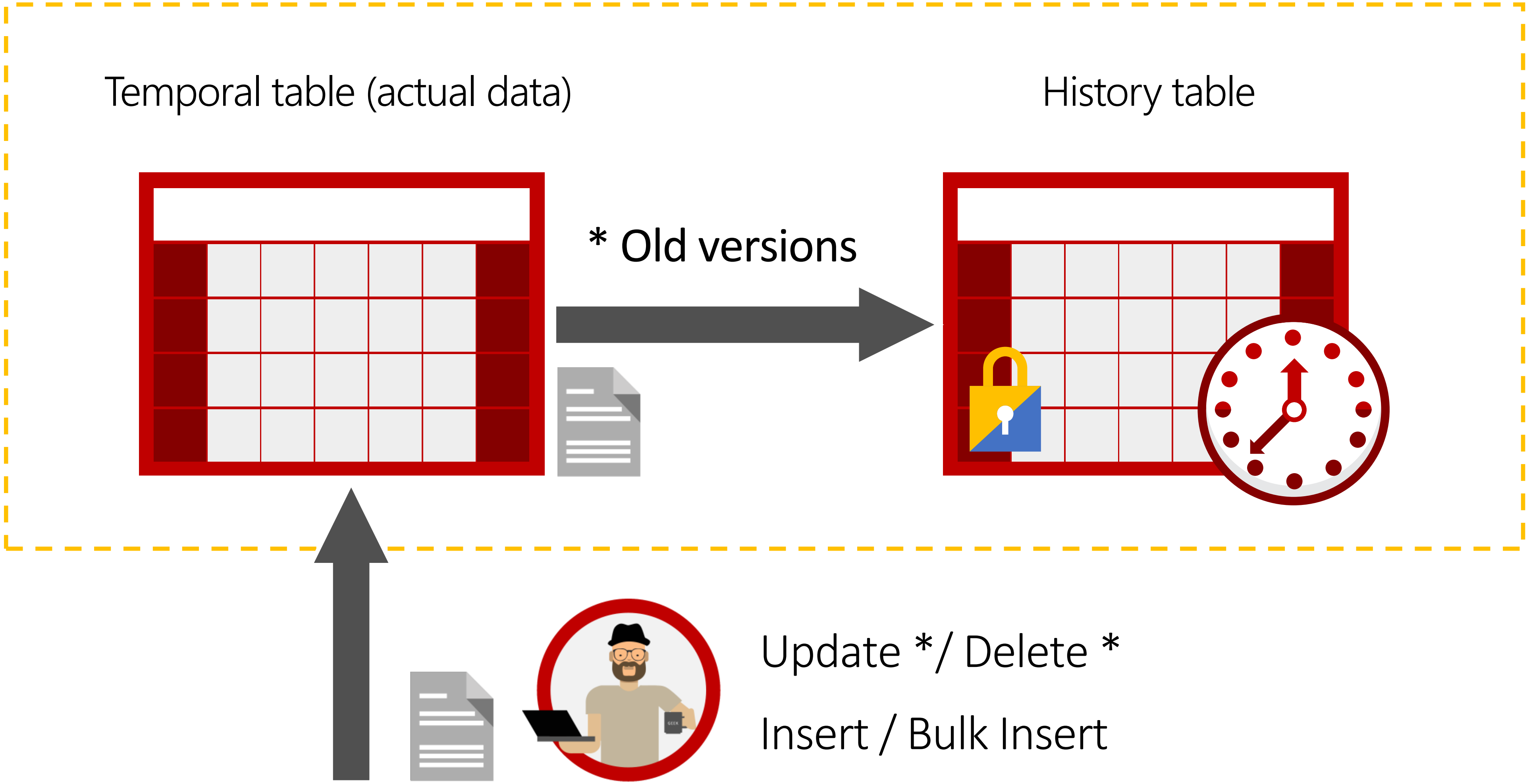
Provides correct information about stored facts at any point in time, or between two points in time

There are two orthogonal sets of scenarios with regards to temporal data:

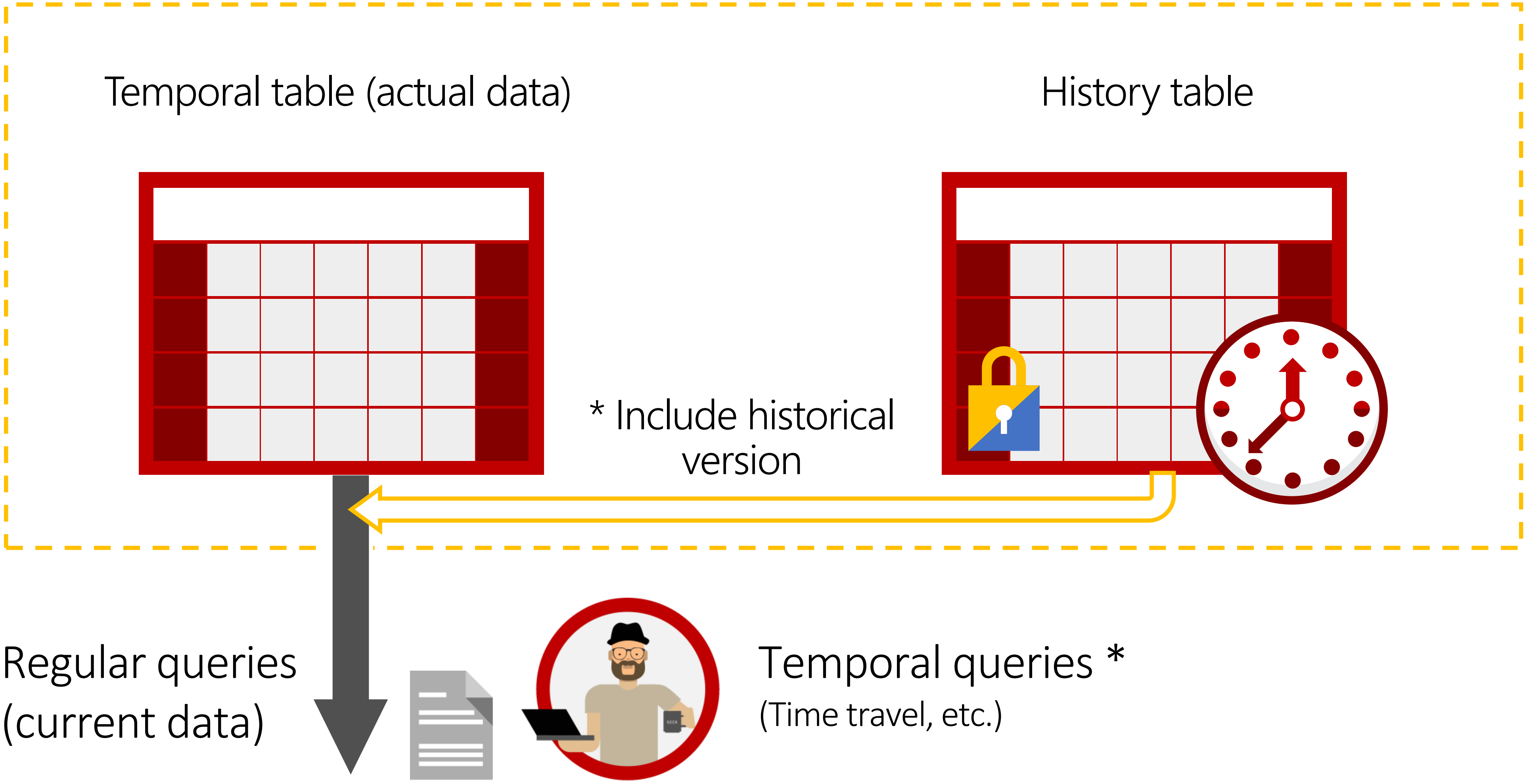
System (transaction)-time

Application-time

How does system-time work?



How does system-time work?



Application-time temporal

```
SELECT * FROM Employee
WHERE VALID_TIME CONTAINS '2013-06-30'

SELECT * FROM Employee
WHERE EmployeeNumber = 1 AND
VALID_TIME OVERLAPS PERIOD ('2013-06-30', '2014-01-01')

/* Temporal join */
SELECT * FROM Employee E
JOIN Position D ON E.Position = D.Position AND
D.VALID_TIME CONTAINS PERIOD E.VALID_TIME
```

Limits of system-time

Time flows 'forward only'

System-time ≠ business-time (sometimes)

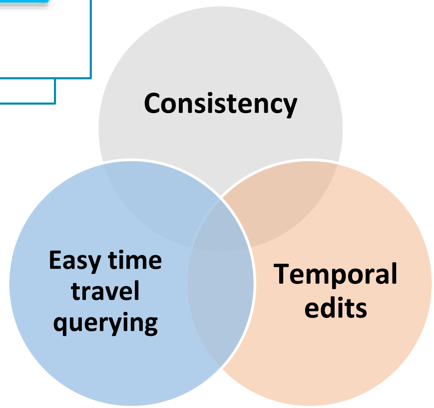
Immutable history, future does not exist

App-time = new scenarios

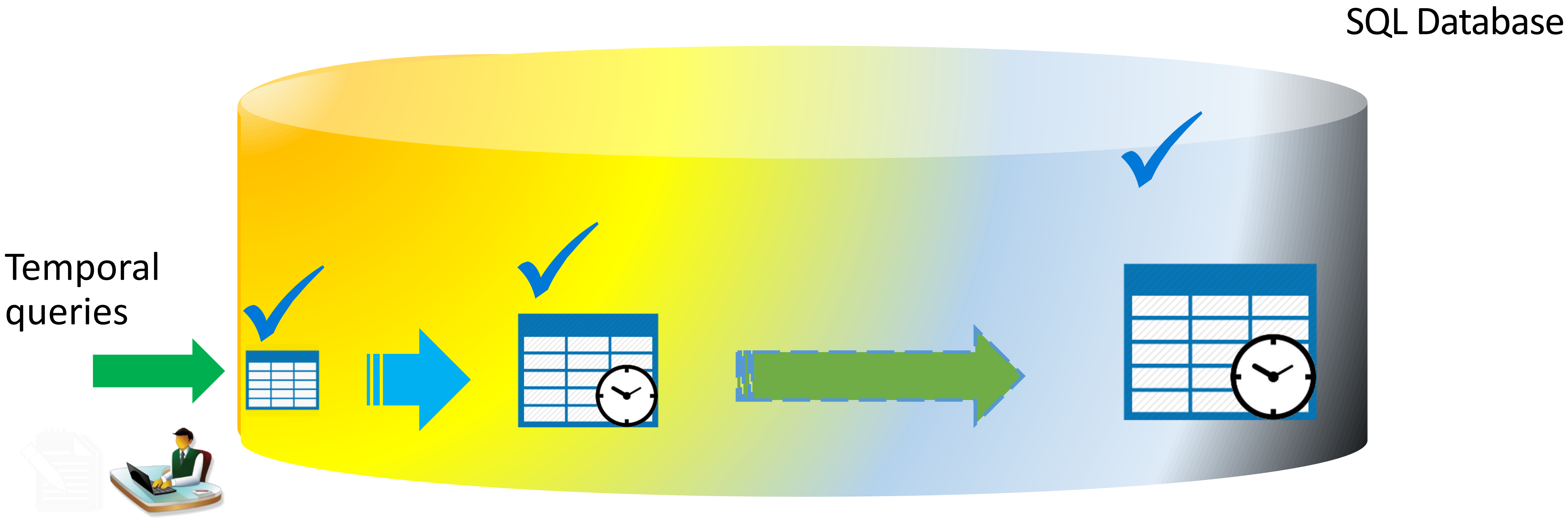
Correct past records as new info is available (HR, CRM, insurance, banking)

Project future events (budgeting, what-if, loan repayment schedule)

Batch DW loading (with delay)



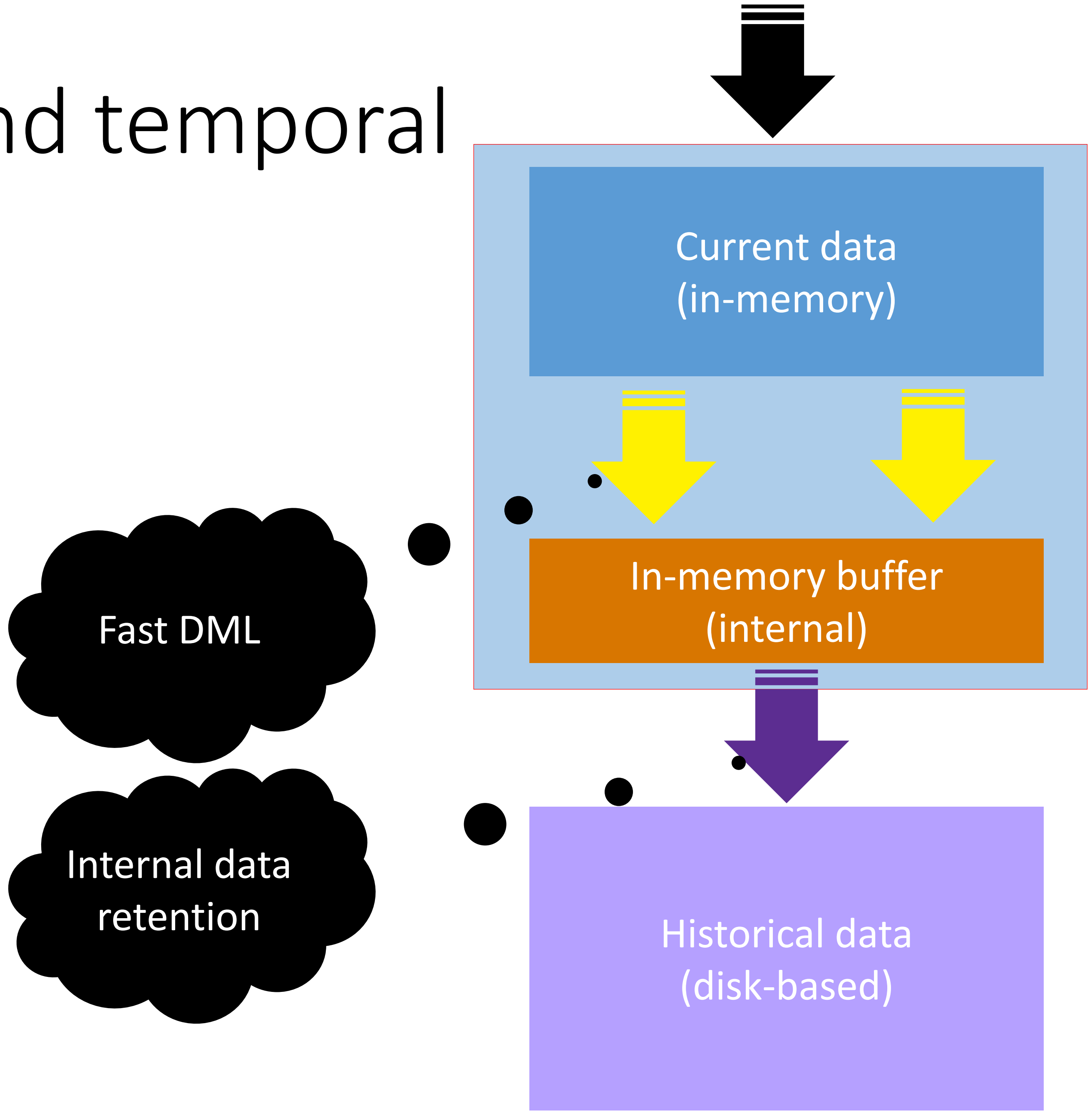
Temporal data continuum



In-Memory OLTP and temporal

Extreme OLTP with cost-effective data history

Disk-based history table
Super-fast DML and current data querying
Temporal querying in interop mode



Summary: Temporal Tables

Quickly add historical versioning with minimal developer effort

Add temporal data to existing tables without downstream impact

Support for temporal queries, auditing, and change tracking

Microsoft