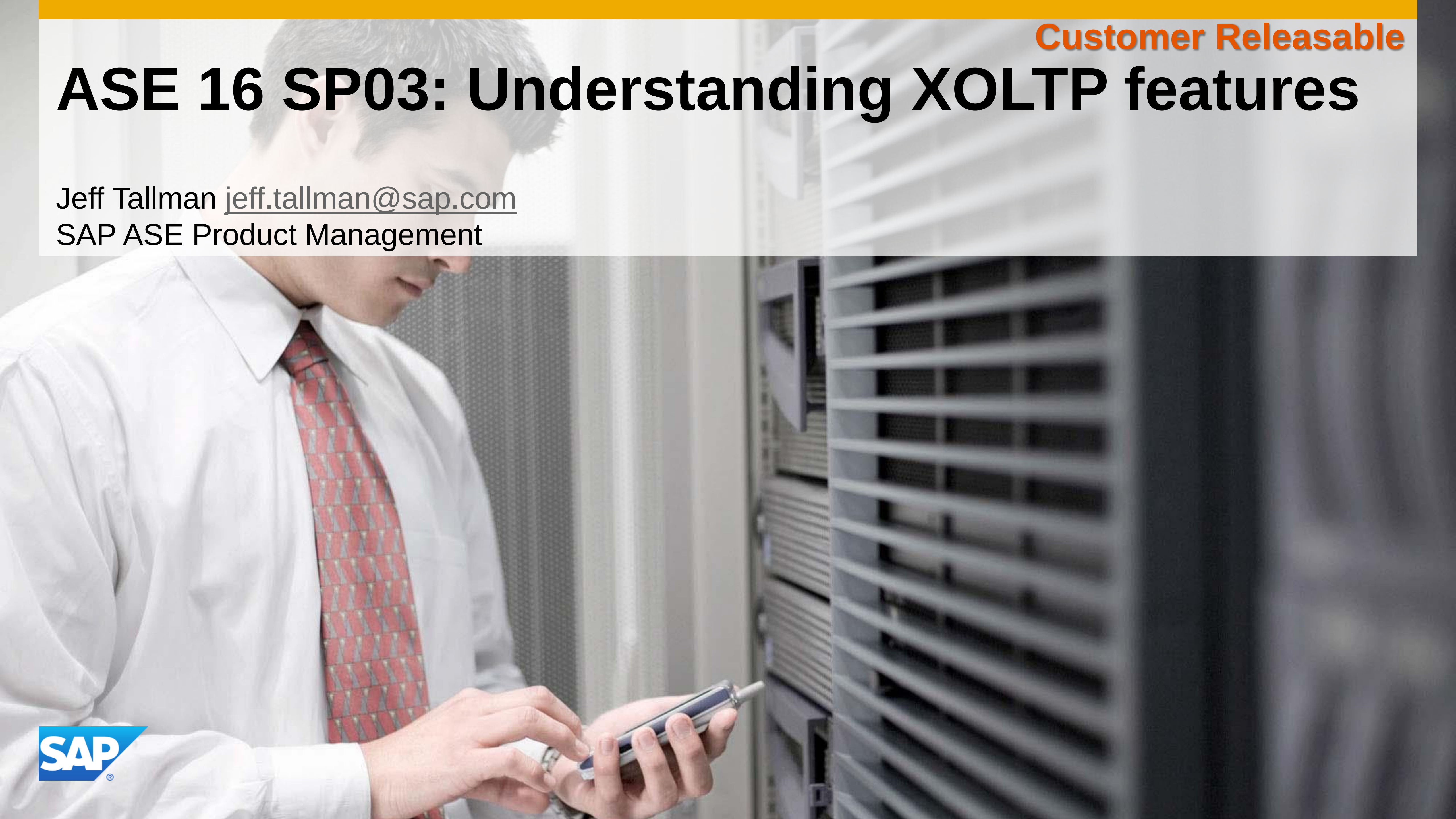
Customer Releasable

ASE 16 SP03: Understanding XOLTP features

Jeff Tallman jeff.tallman@sap.com
SAP ASE Product Management



Disclaimer

This presentation outlines our general product direction and should not be relied on in making a purchase decision. This presentation is not subject to your license agreement or any other agreement with SAP. SAP has no obligation to pursue any course of business outlined in this presentation or to develop or release any functionality mentioned in this presentation. This presentation and SAP's strategy and possible future developments are subject to change and may be changed by SAP at any time for any reason without notice. This document is provided without a warranty of any kind, either express or implied, including but not limited to, the implied warranties of merchantability, fitness for a particular purpose, or non-infringement. SAP assumes no responsibility for errors or omissions in this document, except if such damages were caused by SAP intentionally or grossly negligent.

Agenda

In-Memory Row Store

Hash-Based Index

MVCC

Understanding MemScale

The problem is attempting to solve

Database MemScale & In-Memory Data

What does in-memory mean?

- For most vendors, it is simply running completely in cache – same LIO efforts, etc. – but some tweaking to minimize overhead
- Essentially, we did this with ASE 15.5 with IMDB
  - ✓Schema and static data read from template database on startup
  - ✓In memory data was still in disk optimized storage format
  - ✓Only real optimizations was to cache management (e.g. elimination of MRU/LRU, wash, etc.)

ASE 16 In-Memory Row Store (aka Data Row Cache)

- Focus is XOLTP – *not* analytics
  - ✓In-memory DBMS for analytics….a common use case for DB2 Blue, Oracle Exalytics/In-Memory, etc.
- Data values will be stored independently for memory optimized access
  - ✓e.g. no more parsing of the row ala standard LIO to retrieve values
- MVCC will be used for DRC only to eliminate as much concurrency contention as possible
- DRC will be persisted (unlike IMDB) and have a redo log separate from transaction log
- Focus will be on
  - ✓Frequently read data rows
  - ✓Frequently updated data rows (including initial insert + subsequent updates)
  - ✓May not be any benefit to streaming/bulk inserts in which data is rarely read immediately after inserted

Typical On-Disk Row Storage & Access via B-Tree

We traverse the index B-tree to leaf

- Leaf node contains page # + rid
- Rid essentially is the row number on the page

We fetch the page and look at row offset table on the page

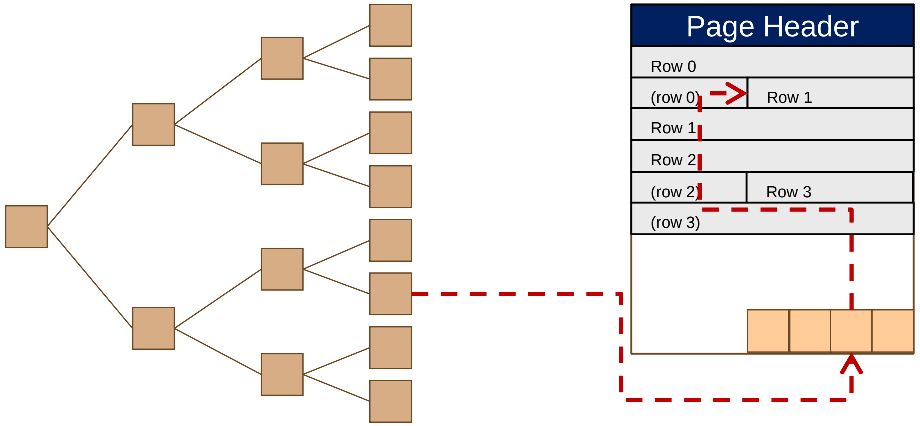
- Do a bufsearch() in cache to find (add) page
- Row offset table is the byte offset within the page that the row begins
- Rows are not necessarily in row offset order
  - ✓ May start that way, but after updates that cause row shrinkage/expansion, etc. the rows might be scattered around on the page

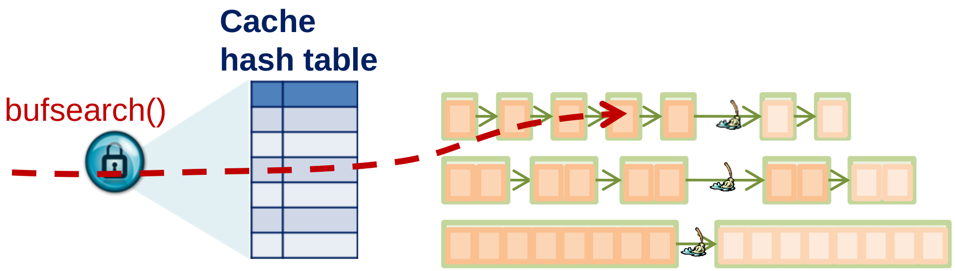
We jump to that byte and read the row

- To find a column, we may have to use the variable length column offset table or compute offset for fixed length columns
- Net result is
  - ✓ Multiple indirection/byte count parsing to read column values from a row of data
  - ✓ Possible high contention on spinlocks (search, mru, etc.)

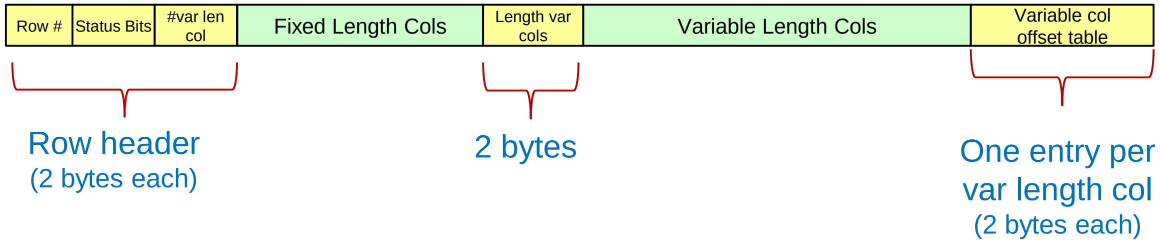
Data modifications have high overhead

- Minimally rewriting the rest of the data row
- Often requires rewriting the page
  - ✓ Has impact on currency as we temporary block other readers/writers (using MASS bit)





Data Row Format



“Disk Optimized Storage”

In-Memory Row Store (A Typical Implementation)

Row consists of in-memory optimized storage (an example)

- Row header includes versioning info
- Each column "cell" is either
  - ✓ a column value (for numerics) (wordsize values)
  - ✓ a pointer to character data allocation
    - – Includes binary()/varbinary(n)

Data Access uses fewer logical reads

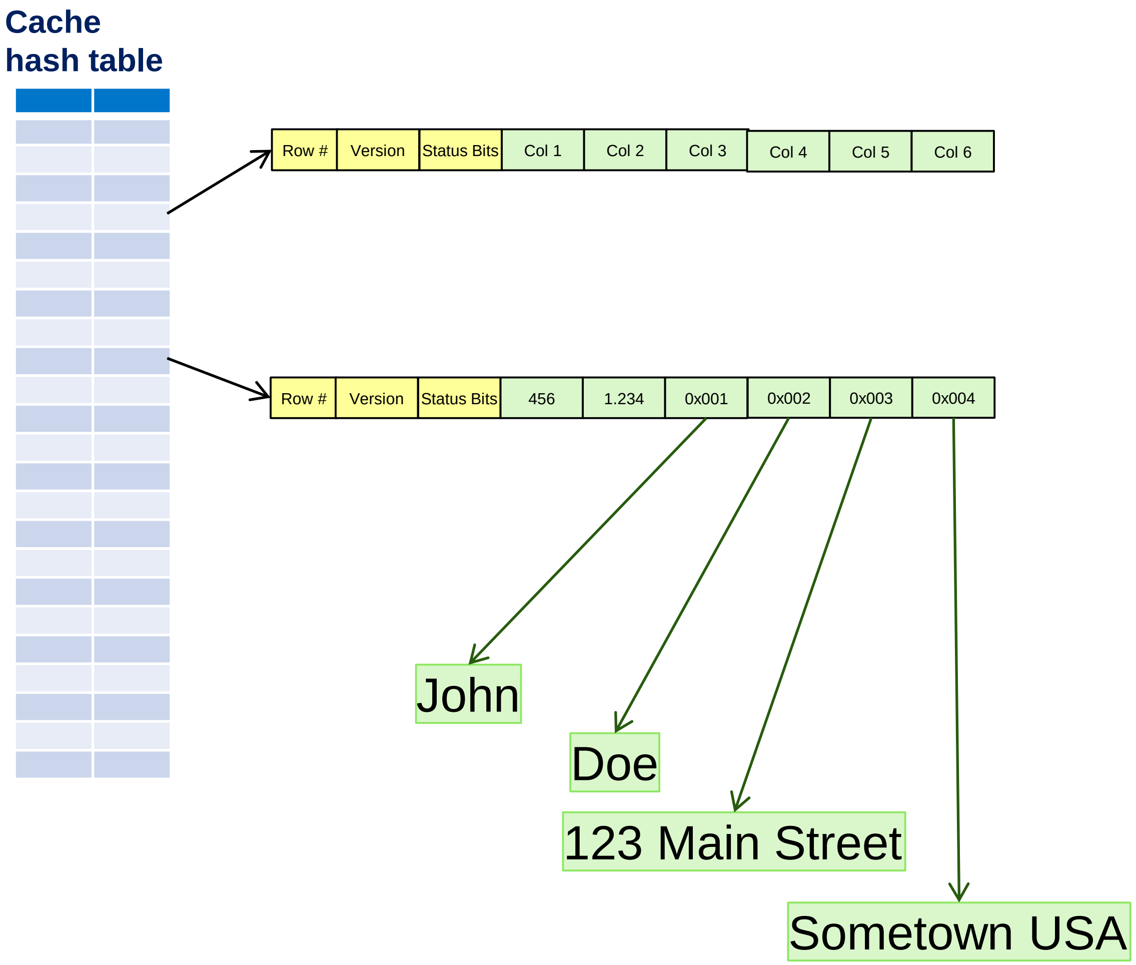
- Exploits hash-based indexing to avoid B-tree traversal

Data Modifications are extremely fast

- Make a copy (version of the row)
- Modify the desired values
- New version linked into cache hash table by using CPU lockless API's

On-Commit Persistence

- When user commits, new versions are written to delta log (typically on SSD)



Row #	Version	Status Bits	Col 1	Col 2	Col 3	Col 4	Col 5	Col 6

Row #	Version	Status Bits	456	1.234	0x001	0x002	0x003	0x004

Where REALLY are the problems??

40-50% is Query Optimization

- Including query execution plan creation
- Other major factors include resource allocation (e.g. proc cache for optimization/sorting or memory for decompression, etc.)

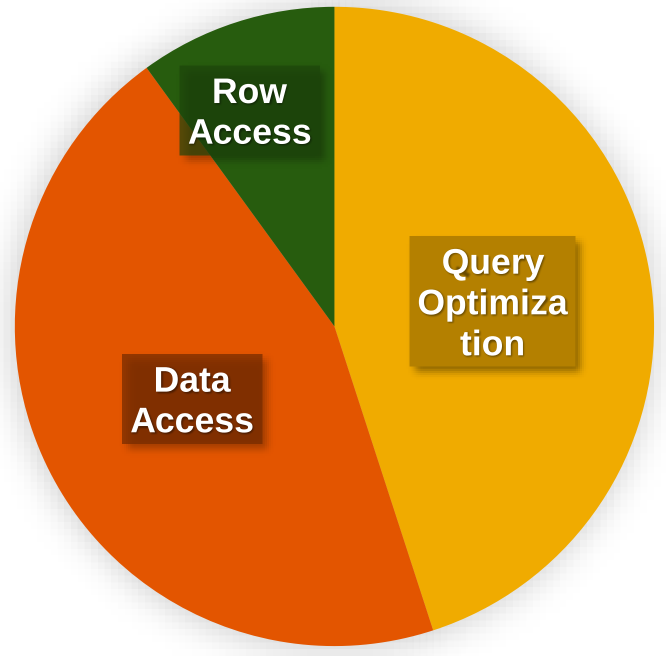
40-50% is DB Access layer

- Although physical IO is most often thought of, the more likely cause for CPU time is resource contention at higher user concurrency
- Other causes are lock/latch acquisition
- DB Access is in reality a function of query optimization or row access
  - ✓ Bad query plans increase cache contention
  - ✓ Inserts need more LIO than single column updates
- Many of the MemScale features address problems here, but it makes sense to FIRST try to “tune out” the obvious (e.g. solve bad QP vs. LLDC)

Only 10-20% is reading the row

- Row/data format parsing

QP Execution Time*



** Percentages are anecdotal averages based on multiple years of QP analysis*

Example: the impact of cache contention on high engine counts

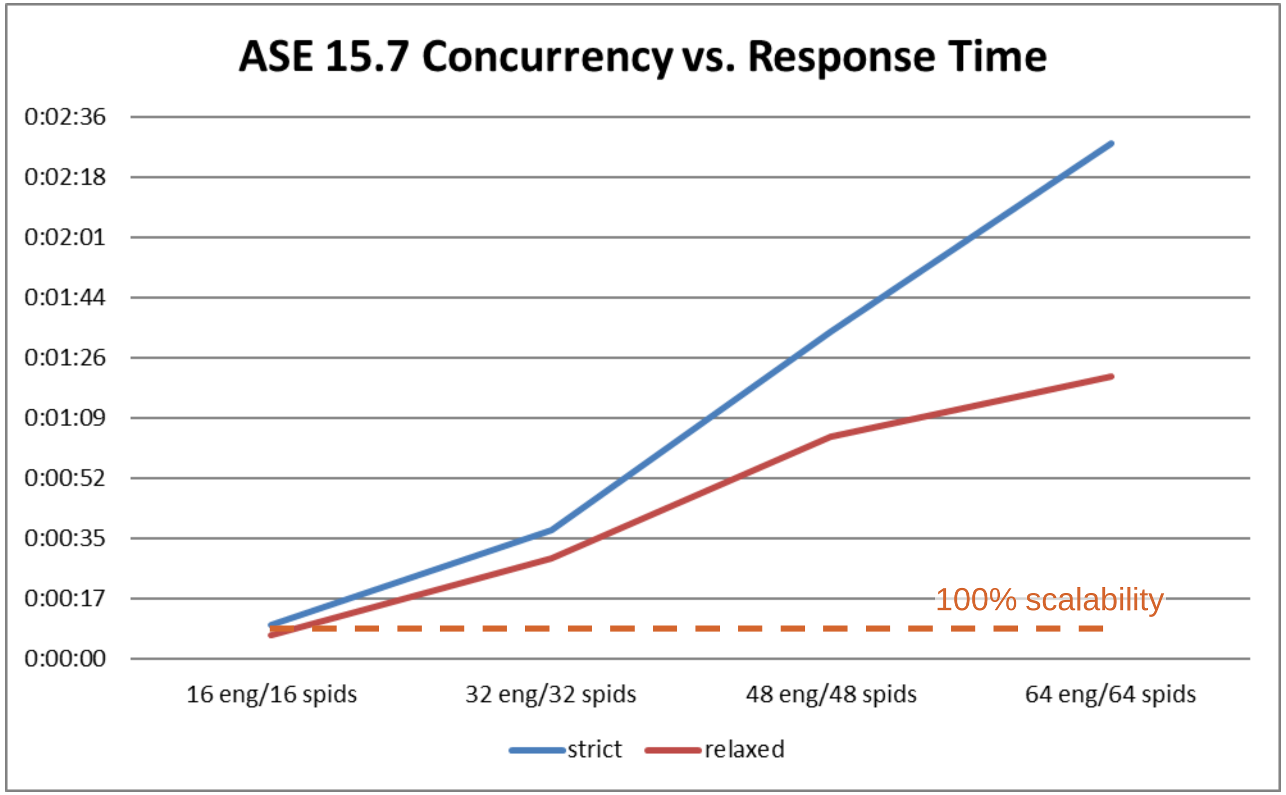
Scenario

- ✓ FSI EMEA customer launches multiple processes all doing the same tasks in parallel
- ✓ Although working on separate accounts, the parallel processes are often after the same data –e.g. the last trade price for a particular stock
- ✓ Typical query batch focuses on 6 key tables
- ✓ Test focused on same number of concurrent tasks as engines – e.g. 16 tasks on 16 engines (baseline)

Results on ASE 15.7

- ✓ As user concurrency and number of engines increase, the response time degrades from 7 seconds to 2m28s (148s) – a 21x degradation!!!
- ✓ Noting that there was considerable data cache spinlock contention – <u>ONE</u> of the 6 key tables was moved to named cache with relaxed cache strategy, *reducing the response time by 1 minute at 64 engines*
  - – At the time, it was considered impractical to move all the tables to a relaxed cache as table volatility may not align with relax cache very well.

Solution → LLDC & MemScale in ASE 16 sp02/sp03



ASE 16 sp02 lock less data cache – eliminate contention when relaxed cache can be used

ASE 16 sp03 IMRS (DRC) – eliminate contention when relaxed cache can't be used (data rows retrieved from DRC while PK index values retrieved from HBC – all are lockless structures)

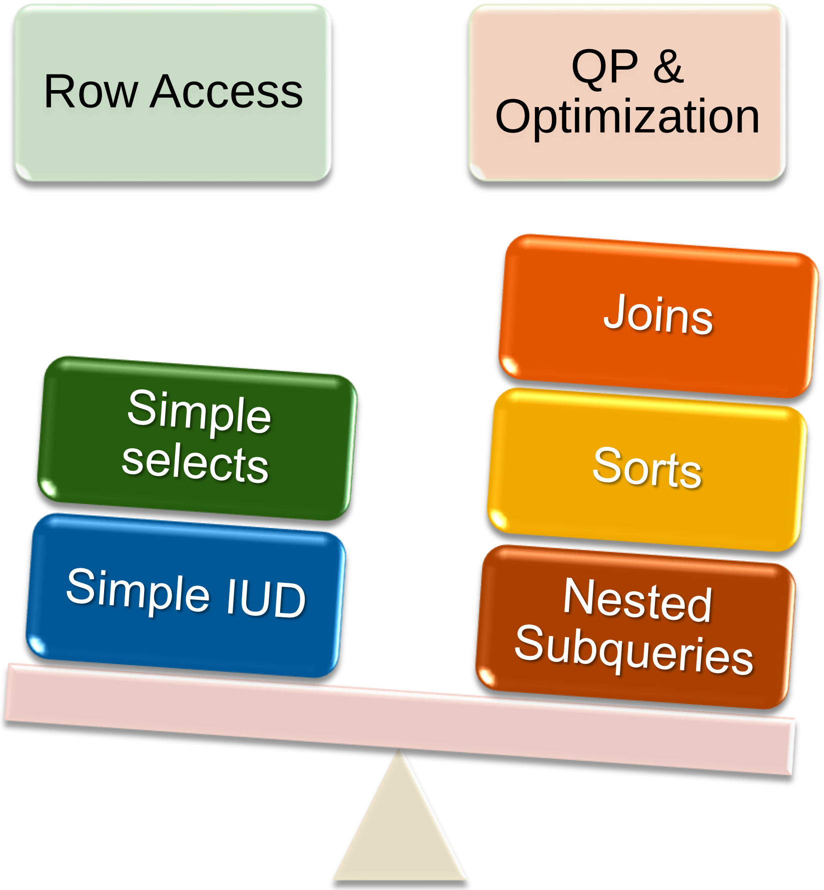
The proportion of time spent obviously varies by query complexity

More complex queries

- More time in optimization
- More LIO increases contention per query
- The effect of SNAP is more pronounced per single query due to exec plan complexity

Simple selects/IUD

- Row access plays a bigger role
- Most of the time is spent in index traversal
  - ✓ For selects this is a single index traversal for pkeys/unique indexes
  - ✓ For IUD, this could be multiple indexes depending on which indexes are impacted by the operation
  - ✓ The more indexes impacted the bigger this cost is as proportion of row access
- IUD – especially atomic IUD – has log semaphore overhead/log commit IO physical IO processing.
- Non-unique indexes could have significant contention on the RID list….the lower the cardinality, the higher the contention probability
- The effect of SNAP is very minimal unless in a tight loop with a lot of iterations where aggregate savings are viable



ASE MemScale drives some changes

Datarows locking

- Necessary for DRC, MVCC, etc.

Threaded kernel & a few extra cores

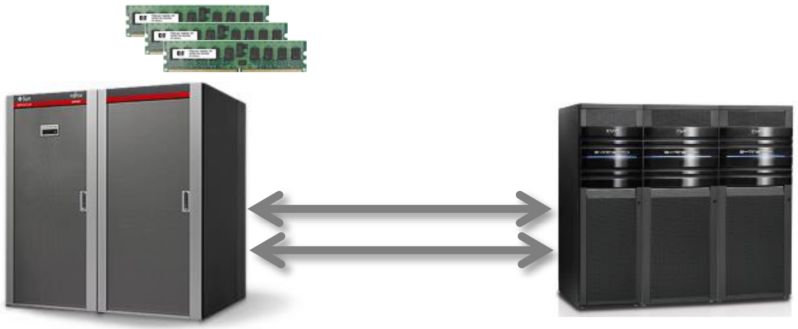
- Necessary for MemScale features
  - ✓PCI configured for compiler engine (needs CPU)
- Necessary for Always-On
  - ✓CI mode native OS threads (need CPU)

Additional Memory

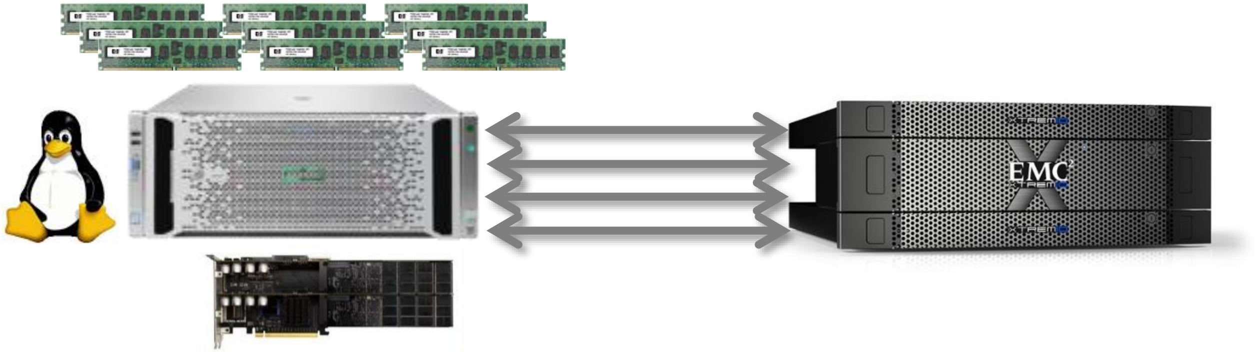
- May be needed to ease implementation of NVCache or IMRS Cache without impacting the default data cache or other named caches used by other databases/applications

PCIe SSD or All Flash Array

- Required for IMRS log devices
- Required for NVCache
- Required for Always-On SPQ



Large virtualized generic hosts with provisioned storage



Smaller dedicated hosts with DBMS HW profile

MemScale Option

Key Thrusts

- Reducing contention
  - ✓ Greater benefits at higher user concurrency/engines due to most likely greater contention
- Reducing CPU & IO Costs
  - ✓ Reduce the number of cpu cycles spent on concurrency controls or memory management

Features:

- (Lockless data cache) (LLDC) → works well for when a relaxed cache simply isn’t enough
- In-Memory Row Store (IMRS) → fills the gap of LLDC by reducing cache contention for any frequent accessed data
- Latch Free B-Tree (LFB) → reduce latch contention on index pages
- Transactional Memory (TSX) (Linux x86 only)→ reduce contention on lock chains & MRU/LRU relinkages
- Simplified Native Access Plans (SNAP) (Linux x86 only) → reduce query execution plan generation cpu time
- NVCache → reduce impact of PIO due to cache flush/re-reads
- Hash Based Cache B-Tree (HBC) → reduce Pkey/Unique index traversal/cache contention
- MVCC → reduce lock contention between readers/writers
- IMDB → eliminate unnecessary physical reads for nondurable data

Separate features, but often complimentary for different application modules

- Key requirements: threaded kernel + datarows locking + PCIe or AFA SSD storage (IMRS, NVCache)

DRC (aka IMRS)

Data Row Cache/In-Memory Row Store

In Memory Row Store Tables (DRC) (1)

Key point to remember

- All data eventually is on page-based disk

In-memory row store cache

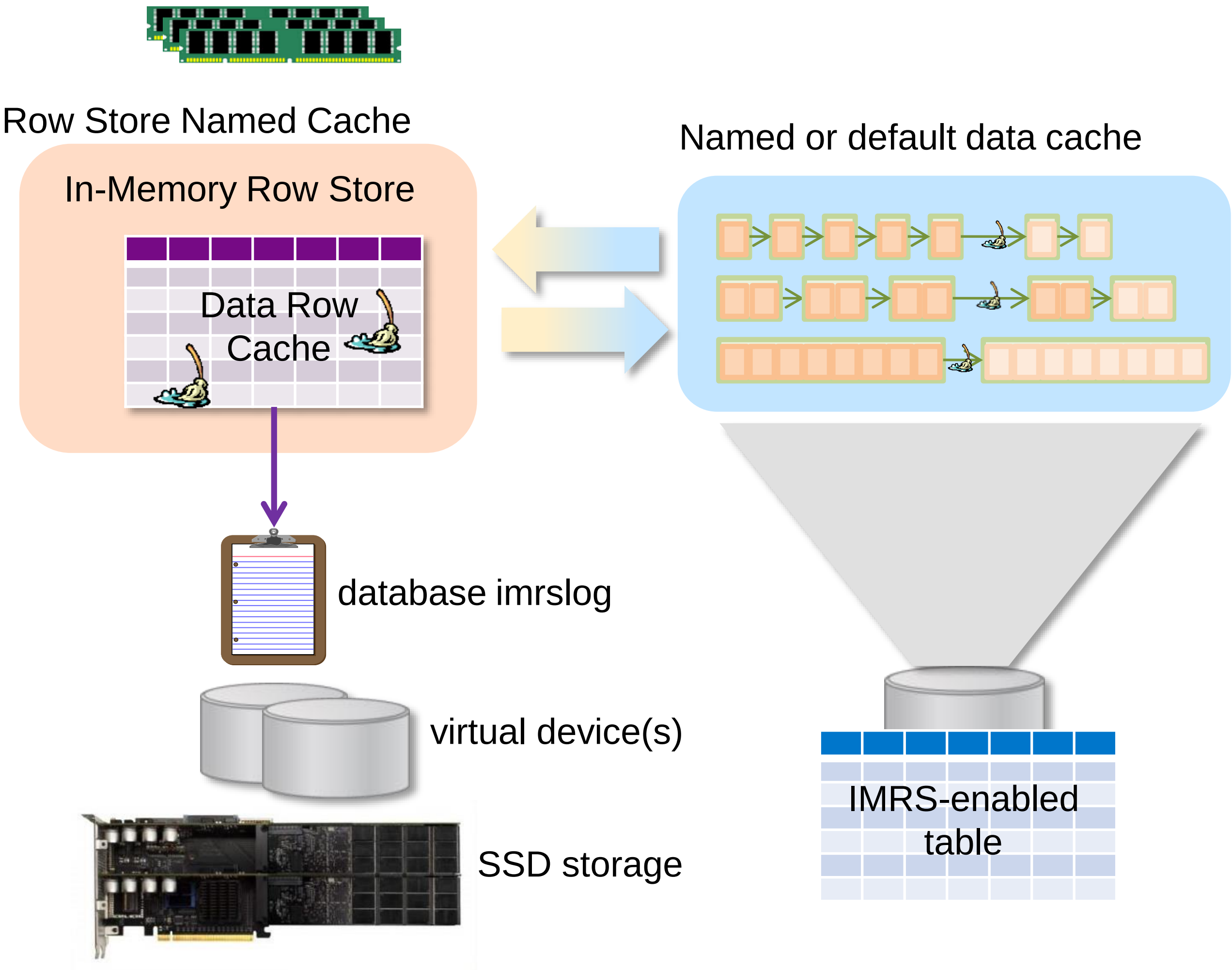
- A special type of named cache that is used as a row store buffer
- 1:1 correspondence with a database

Data Row Cache (DRC)

- The cached rows from imrs-enabled tables

Database imrslog

- A database device/segment that contains both the imrs cached rows as well as the committed txn journal
  - ✓ Sysimrslogs (similar to syslogs) plus data
- It is created on a standard ASE virtual device



In Memory Row Store Tables (DRC) → The Advantages

Eliminates LIO/cache contention on frequently accessed data

- A bit better than LLDC as it works irrespective of relaxed cache strategy

Eliminates performance hit of compression

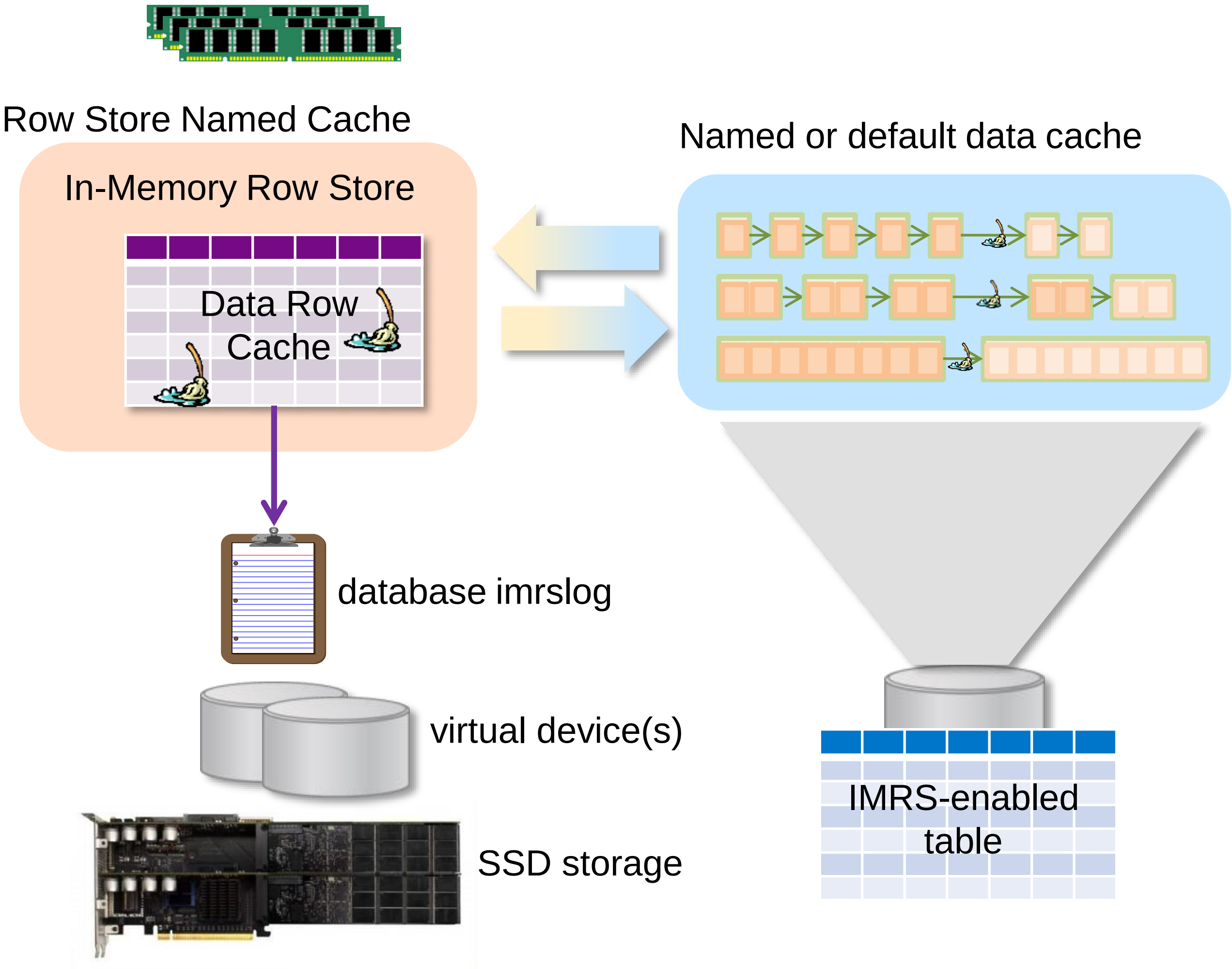
- Data in DRC is uncompressed until packed back to page buffers

Avoids penalty of row forwarding

- When row is packed back to page buffer, it may cause row forwarding, but access while in DRC bypasses any issues.

Concurrent commits to IMRSLOG

- Concurrent txns can commit to IMRS log simultanteusly, reducing ULC usage/log semaphore contention to just index insertions



In Memory Row Store Tables: Garbage Collection

IMRS uses a number of ASE Tasks

- IMRS GC
- LOG GC
- IMRS PACK
- HBC GC

IMRS GC (Garbage Collectors)

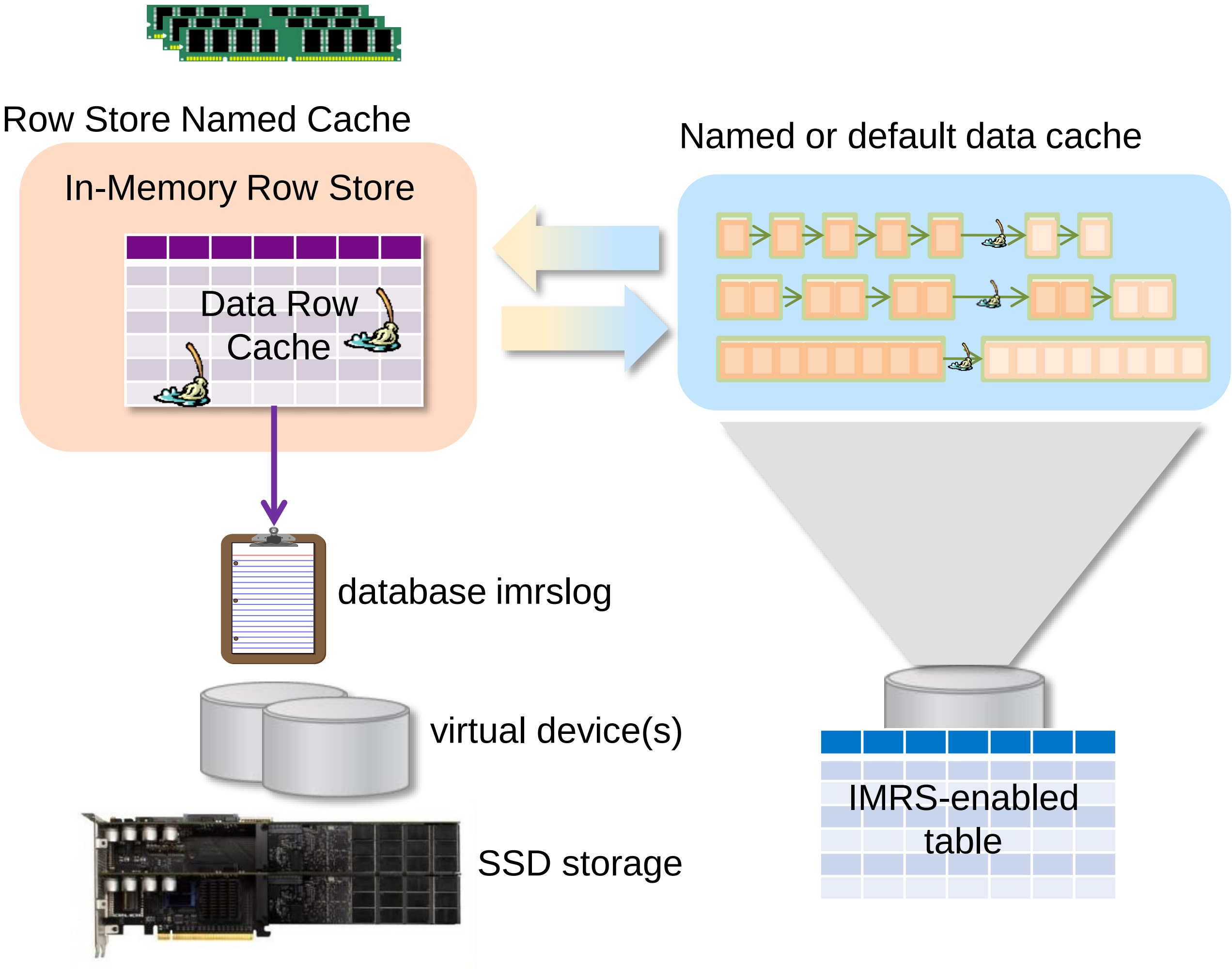
- Threads that clean DRC by throwing away old versions of rows and cleaning deleted rows
- Separate tasks handles LOB columns
- Default is 2 of each

IMRS PACK

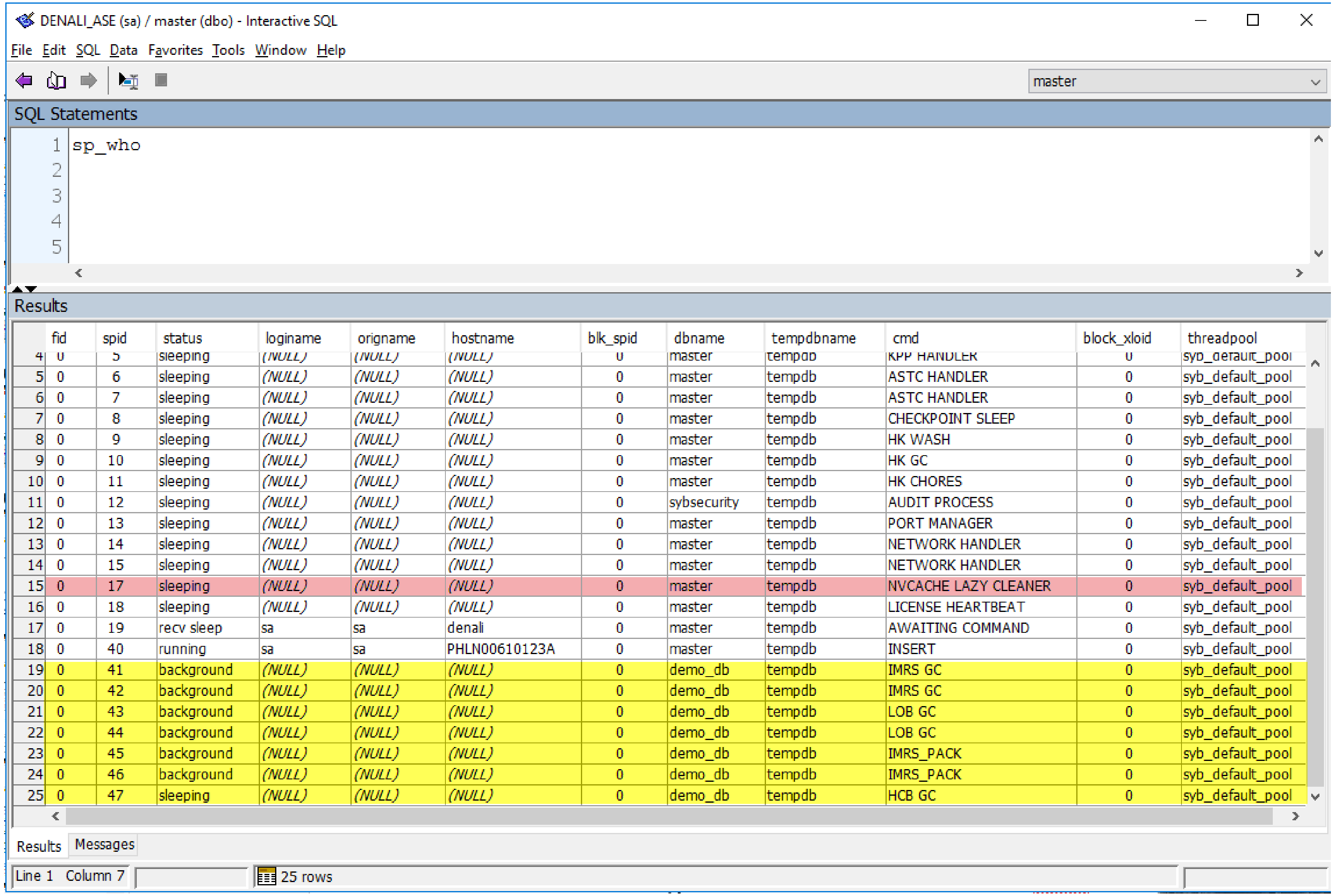
- Threads that make room in the IMRS by packing colder data back into the page buffers
- Default is 2

HBC GC (Garbage Collection)

- Removes cached index rows from hash-based cache for rows no longer in IMRS



IMRS & NVCache ASE Tasks (spids)


DENALI_ASE (sa) / master (dbo) - Interactive SQL
File Edit SQL Data Favorites Tools Window Help
master
SQL Statements
Results
Results Messages
Line 1 Column 7
25 rows

```
sp_who
```

	fid	spid	status	loginame	origname	hostname	blk_spid	dbname	tempdbname	cmd	block_xloid	threadpool
4	0	5	sleeping	(NULL)	(NULL)	(NULL)	0	master	tempdb	KPP HANDLER	0	syb_default_pool
5	0	6	sleeping	(NULL)	(NULL)	(NULL)	0	master	tempdb	ASTC HANDLER	0	syb_default_pool
6	0	7	sleeping	(NULL)	(NULL)	(NULL)	0	master	tempdb	ASTC HANDLER	0	syb_default_pool
7	0	8	sleeping	(NULL)	(NULL)	(NULL)	0	master	tempdb	CHECKPOINT SLEEP	0	syb_default_pool
8	0	9	sleeping	(NULL)	(NULL)	(NULL)	0	master	tempdb	HK WASH	0	syb_default_pool
9	0	10	sleeping	(NULL)	(NULL)	(NULL)	0	master	tempdb	HK GC	0	syb_default_pool
10	0	11	sleeping	(NULL)	(NULL)	(NULL)	0	master	tempdb	HK CHORES	0	syb_default_pool
11	0	12	sleeping	(NULL)	(NULL)	(NULL)	0	sybsecurity	tempdb	AUDIT PROCESS	0	syb_default_pool
12	0	13	sleeping	(NULL)	(NULL)	(NULL)	0	master	tempdb	PORT MANAGER	0	syb_default_pool
13	0	14	sleeping	(NULL)	(NULL)	(NULL)	0	master	tempdb	NETWORK HANDLER	0	syb_default_pool
14	0	15	sleeping	(NULL)	(NULL)	(NULL)	0	master	tempdb	NETWORK HANDLER	0	syb_default_pool
15	0	17	sleeping	(NULL)	(NULL)	(NULL)	0	master	tempdb	NVCACHE LAZY CLEANER	0	syb_default_pool
16	0	18	sleeping	(NULL)	(NULL)	(NULL)	0	master	tempdb	LICENSE HEARTBEAT	0	syb_default_pool
17	0	19	recv sleep	sa	sa	denali	0	master	tempdb	AWAITING COMMAND	0	syb_default_pool
18	0	40	running	sa	sa	PHLN00610123A	0	master	tempdb	INSERT	0	syb_default_pool
19	0	41	background	(NULL)	(NULL)	(NULL)	0	demo_db	tempdb	IMRS GC	0	syb_default_pool
20	0	42	background	(NULL)	(NULL)	(NULL)	0	demo_db	tempdb	IMRS GC	0	syb_default_pool
21	0	43	background	(NULL)	(NULL)	(NULL)	0	demo_db	tempdb	LOB GC	0	syb_default_pool
22	0	44	background	(NULL)	(NULL)	(NULL)	0	demo_db	tempdb	LOB GC	0	syb_default_pool
23	0	45	background	(NULL)	(NULL)	(NULL)	0	demo_db	tempdb	IMRS_PACK	0	syb_default_pool
24	0	46	background	(NULL)	(NULL)	(NULL)	0	demo_db	tempdb	IMRS_PACK	0	syb_default_pool
25	0	47	sleeping	(NULL)	(NULL)	(NULL)	0	demo_db	tempdb	HCB GC	0	syb_default_pool

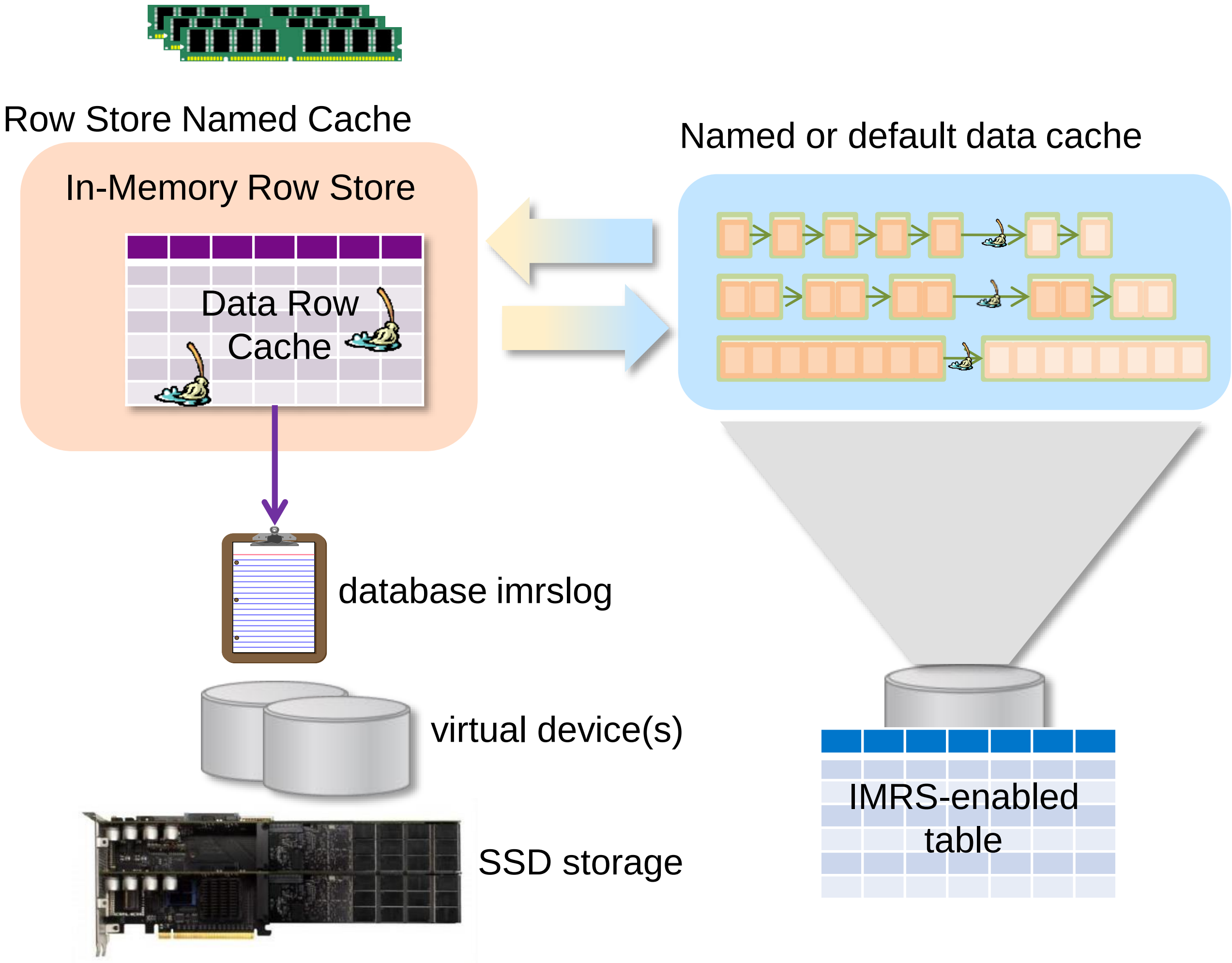
In Memory Row Store Tables (DRC) (3)

IMRS tables

- Must be datarows locked
- Cannot be from tempdb tables
- Frequently accessed rows from DRC-enabled tables will automatically be cached
  - ✓Only data rows are cached - not index rows
- Rows are 'packed' back into disk page layout when aged out of cache

Comments on temp tables

- By default, they are allpages locked vs. datarows
- Select/into uses large IO pools/bulk inserts which bypasses IMRS
- Not frequently accessed - often read once & done
- If caching desired, use IMDB tempdb

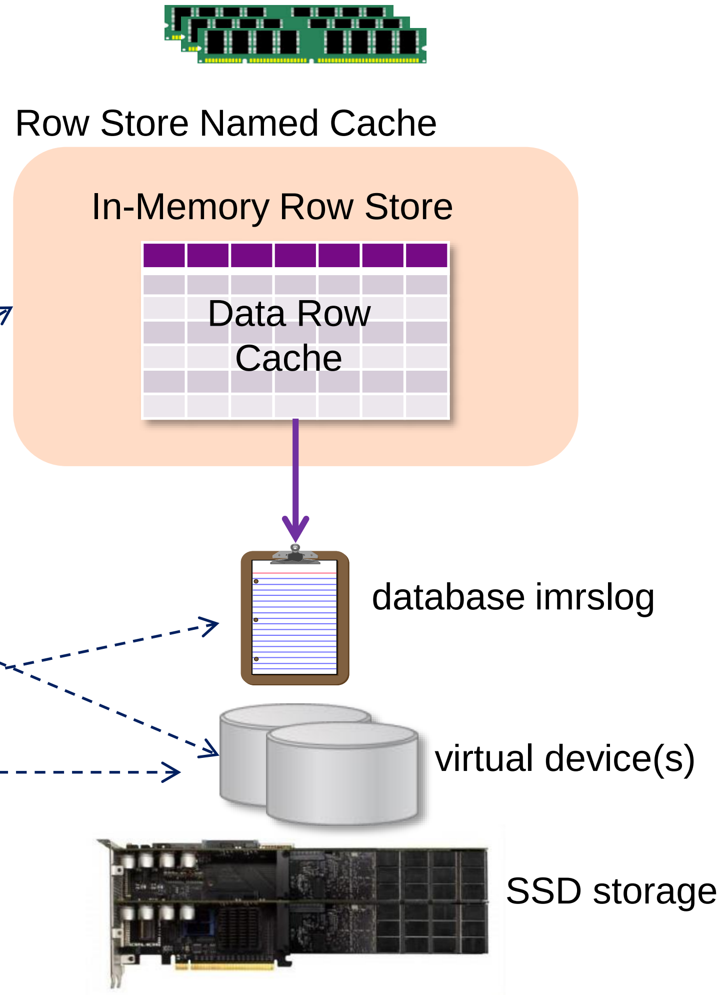


In Memory Row Store Pieces & Parts

```
sp_cacheconfig [ cache_name ]
    [, “ cache_size [ P | K | M | G ]” ]
    [, logonly | mixed | inmemory_storage | row_storage ]
```

```
disk init name = ‘device_name’,
           physname = “physical_name”,
           size = “size_specifier”,
           type = “imrslog”
```

```
create database dbname
     [on {default | database_device [, ...] } ]
     [log on database_device [,...] ]
     [imrslog on database_device [= size] [, ...]]
     [row storage on cache_name]
[with row_caching { ON | OFF }
   [[,] snapshot_isolation { ON | OFF } ]
   [ other sub-clauses ... ]
]
…
```



Syntax

Create/alter database

- You need to specify both the imrslog device(s) and the row storage cache name
  - ✓This is the only association between the cache and the imrslog devices - e.g.
  - ✓This differs from IMDB in which the db device is created from the cache (see next slide)
- …with row_caching { ON | OFF }
  - ✓Database wide default setting
  - ✓If enabled, future tables created will inherit row caching.
  - ✓For alter database, previously created tables do not inherit setting
    - You will need to individually alter desired tables to modify, or…
    - …specify “FOR ALL TABLES” to modify existing tables to enable DRC

Create/alter table

- Enables for the table - any rows in table
  - ✓Alter table …set row_caching off disables temporarily but existing rows in IMRS are still cached
  - ✓Alter table …set row storage off disables completely

```
[create | alter] database dbname
     [on {default | database_device [, ...] } ]
     [log on database_device [,...] ]
     [imrslog on database_device [= size] [, ...]]
     [row storage { on cache_name | off }]
[with row_caching { ON | OFF }
     [[,] snapshot_isolation { ON | OFF } ]
   [ other sub-clauses ... ]
]
…
create table tabname ( <column-list> )
  lock datarows
  [with row_caching { on | off }
   [[,] snapshot_isolation { on | off }]]
…

alter table tabname          -- on = enabled
set row_caching [ on | off ] -- off = temporary off
…

-- set row_caching off first
alter table tabname
set row storage off   -- permanent
…
```

IMRS Notes for create/alter database

Sizing

- Start with 2x the log size for the cache size
  - ✓Minimum size you should start with is 2GB
  - ✓Cache overhead is 250MB
- Note that the imrslog device is a log similar to the txn log
  - ✓So the imrslog device will need to be considerably larger than the cache size (5x to start??)

If you need to increase

- Use disk resize to increase the size of the imrslog device
- Use sp_cacheconfig to increase the size of the IMRS
- Use alter database to increase the size of the imrslog

The following commands are not applicable

- Mirroring (deprecated in ASE anyhow) → DISK MIRROR, UNMMIRROR, REMIRROR
- Disk/device recovery in master → DISK REFIT, DISK REINIT
  - ✓So make sure master is backed up after any IMRS configuration change!!

Sizing IMRS……decisions….decisions….

A twist – data in IMRS doesn’t have to be in page cache

Soooo….which do you need?

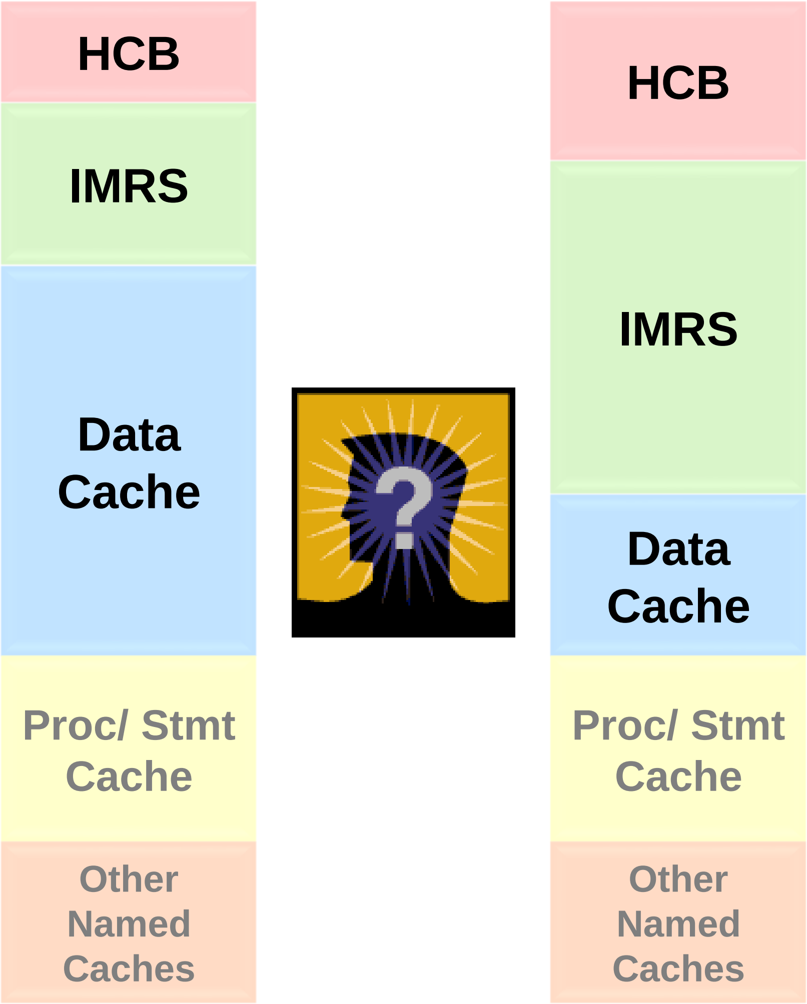
- Larger data cache or larger IMRS+HCB????

Answer: “depends” of course

- Larger IMRS may give more benefit – especially on less memory constrained systems

Some considerations

- Data cache used by indexes
  - ✓Check monCachedObject
- Space for data just read before escalating to DRC
- Space for data being packed out of DRC
- Monitor monOpenObjectActivity & monCachedObject
  - ✓If you see high LIO on a table, consider increasing IMRS size



HCB
IMRS
Data Cache
Proc/ Stmt Cache
Other Named Caches

HCB
IMRS
Data Cache
Proc/ Stmt Cache
Other Named Caches

Is there a penalty??

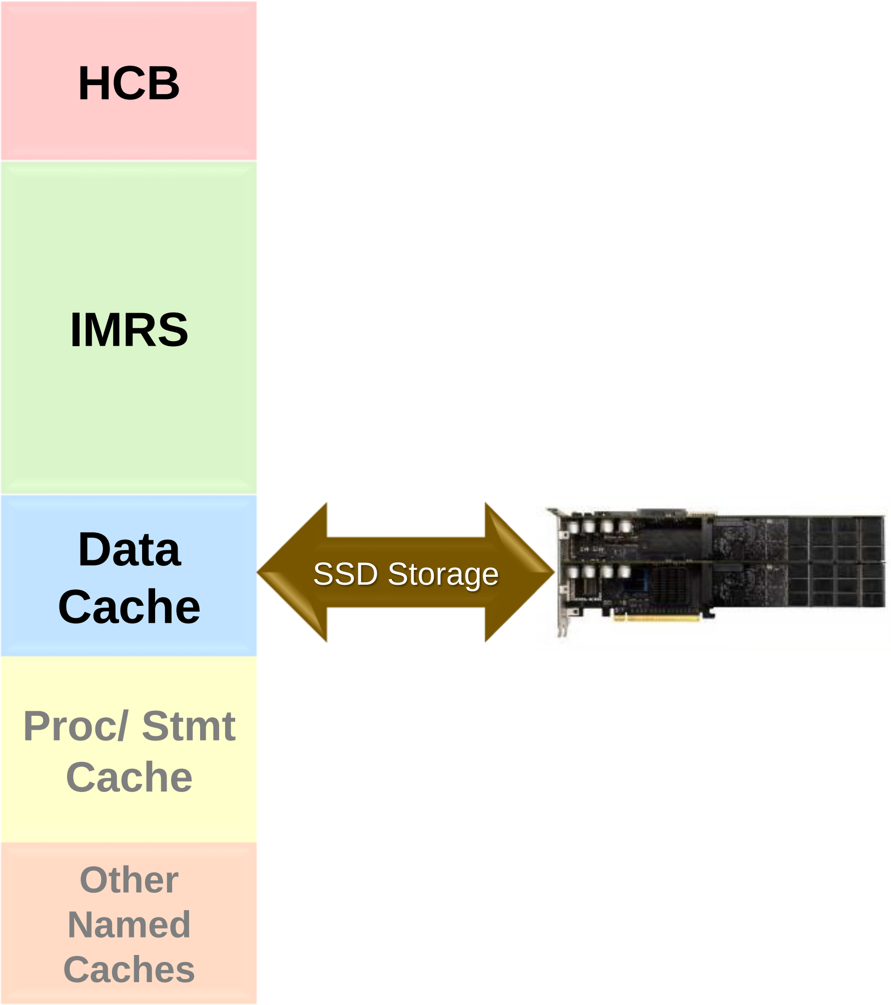
Ummmm…yeah… "it depends" ≈ "there are tradeoffs"

Packing may require Physical Reads

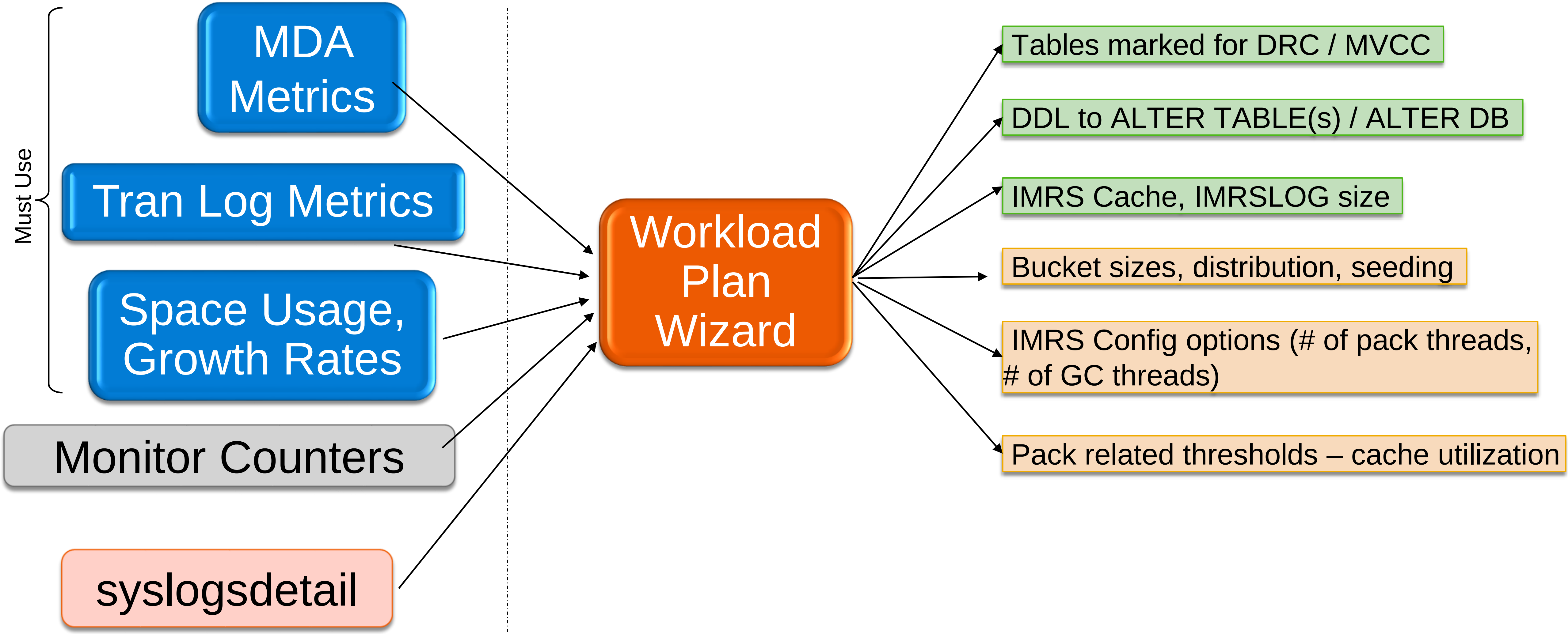
- If the page has been removed from page cache, when the row is packed back to page, it may require a physical read
- This slows the packing process – which could impact DRC caching as memory may not be available
- This may have an impact on data cache as physical read will likely bump some other page out of cache

The solution: SSD storage (Jeff's comments)

- NVCache wouldn't work as it would incompatible with IMRS
  - ✓Requires binding the entire DB to the NVCache



Avoiding the guessing….. Workload Profiler



Must Use
MDA Metrics
Tran Log Metrics
Space Usage, Growth Rates
Monitor Counters
syslogsdetail
Workload Plan Wizard
Tables marked for DRC / MVCC
DDL to ALTER TABLE(s) / ALTER DB
IMRS Cache, IMRSLOG size
Bucket sizes, distribution, seeding
IMRS Config options (# of pack threads, # of GC threads)
Pack related thresholds – cache utilization

Plus some help from autotuning

Sp_dboption ‘auto imrs partition tuning’

- IMRS PACK threads consider a number of factors such as reuse rates, growth rates, memory usage, etc. to determine to selectively disable DRC for a table/partition
- If turned off, already disabled tables remain disabled

Jeff’s comments/observations

- You may want to disable in testing – e.g. if all you are testing is the insert portion of your app, the auto tuning might just disable it as the data isn’t being read…

```
00:0007:00000:00042:2017/03/22 14:21:58.31 server   Disabled IMRS use for (dbid = '6', object id =
'1872006669', partition id = '1872006669' row_type = Inserted) due to lack of reuse of rows in IMRS.
```

Disabling is a two step process..plus two more to clean up

Turn off the DRC - this will repack the rows

- Sp_dboption <dbname>, single, true
- Alter database <dbname> row storage off

Disable the IMRSlog

- Alter database <dbname> imrslog off
- Sp_dboption <dbname>, single, false

Removing the cache

- Sp_cacheconfig <imrscache>,’0’

Removing the IRMSlog device

- Sp_dropdevice <imrsdevice>

Decreasing IMRS cache is a bit tricky

Problem

- On recovery, we process imrslog.
- As we process the imrslog, we recache committed rows
- If the IMRS cache is too small, recovery fails

Fail Safe Technique

- Force a pack of the IMRS cache
  - ✓Entire database → Alter database <dbname> set row storage off
  - ✓Table level → alter table <tablename> set row_caching off -- (temporary)
- Shutdown ASE
- Alter IMRS cache size to the new size
- Restart ASE
- Re-enable row_caching (database level)
  - ✓If table level and only temporarily disabled, on reboot, it may be reset automatically (needs verified)

Backup and Recovery

IMRSLOG has ‘on commit’ persistence

- When txn commits, rows modified in DRC are flushed to imrslog
- If txn rollsback, DRC versions are rolled back with no writing to imrslog
- Rows promoted from page buffer cache to DRC are not logged
  - ✓Therefore on recovery, the cache state is not recovered

Backup

- Full database backups will back up the imrslog
- Cumulative database dumps are not supported with databases using IMRS
- Transaction log dumps includes imrslog

Recovery

- Database is recovered like normal
- After tranlog is recovered, imrslog is applied and rows are re-cached in IMRS

DML Operations & DRC

Normal DML operations

- Insert →
  - ✓ Atomic row and insert/select (non bulk) is inserted into DRC only
  - ✓ Insert/select (bulk) and merge (bulk) bypass DRC and insert into page
- Select → depends
- Update → depends
- Delete
  - ✓ If in DRC, row is removed from DRC
  - ✓ Row is deleted from page cache

Bulk inserts

- Fast bcp → bypasses DRC
- Fast bcp w/ indexes → bypasses DRC
- Slow bcp → inserts into DRC
  - ✓ To avoid, you need to use --init_string=‘set row_caching off’

SQL	Cache
Inserts	DRC
Insert/select	DRC
Insert/select (ins_by_bulk)	page
Infrequently queried rows	page
Frequently queried rows	DRC
Infrequently updated rows	page
Frequently updated rows	DRC
Deletes	Page & DRC

IMRS Operations: LOBs & Triggers

In-row LOBs are in DRC

- Selects & updates to text column are made in the DRC vs. page cache

Off-row LOBs (normal text chain) are not in DRC

- The other columns would be
- Access to the text column would require normal page buffer reads

Triggers

- Triggers are supported on IMRS enabled tables
- Inserted & deleted tables are created from
  - ✓Syslogs for non-IMRS cached rows
  - ✓Cached versions of IMRS enabled table rows

IMRS & cache operations

Data is first read into the page cache

- Only after a row is accessed repeated is the row moved to IMRS
- Once the row is in the IMRS, there is no further need for page in page cache
  - ✓ We don’t explicitly flush it, but it isn’t pinned either, so if it gets replaced on MRU, it gets replaced.
  - ✓ If page is no longer in cache, it will need to be re-read for several operations
    - Packing row back onto page
    - Deletes of row

IMRS GC (Garbage Collector), IMRC_PACK, LOG GC & HCB GC

- Rows in IMRS not frequently accessed are repacked to page buffer by IMRS_PACK
- Once row is repacked or if a row is deleted or as MVCC versions are released, the space is reclaimed by IMRS GC
- LOB GC is used to remove space consumed by off-row LOB versions which are in the page store
- HCB GC does the same for Hash Cache B-tree index entries

IMRSLOG & tuning considerations

For filesystem devices, use dsync=true/directio=false

- Writes to the log can be concurrent (unlike syslogs), but…..
  - ✓Sysimrslogs isn't exactly like syslogs – we try to keep each txns records in close proximity vs. totally serialized.   This allows concurrent commits (within reason).
- We want the commit to return as fast as possible
  - ✓If we used async IO, task would sleep pending IO polling and completion
  - ✓By using a synchronous/blocking IO, task completes as soon as IO completes

IMRSLOG & space allocation

- Normally in the traditional transaction log, once a log page fills, subsequent log writes wait for log pages to be allocated.
- To avoid this overhead, it appears that imrslog preallocates pages via sp_configure
  - ✓Likely to support concurrent writes and to allow all the transaction records to be more collocated.
  - ✓sysimrslogs prealloc percent → defaults to 50%
  - ✓sysimrslogs prealloc size → defaults to 65536 memory pages (2K) (or 128MB)

Hash Cached B-Tree (HCB)

Support for hash indexing

There are 3 common index formats

B-Tree (most common)

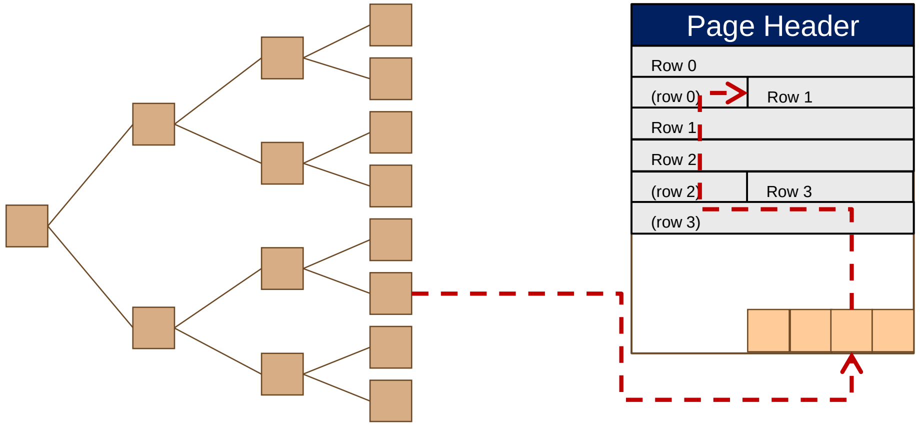
- Typical tree structure
- Nodes are composed of index key values + RID

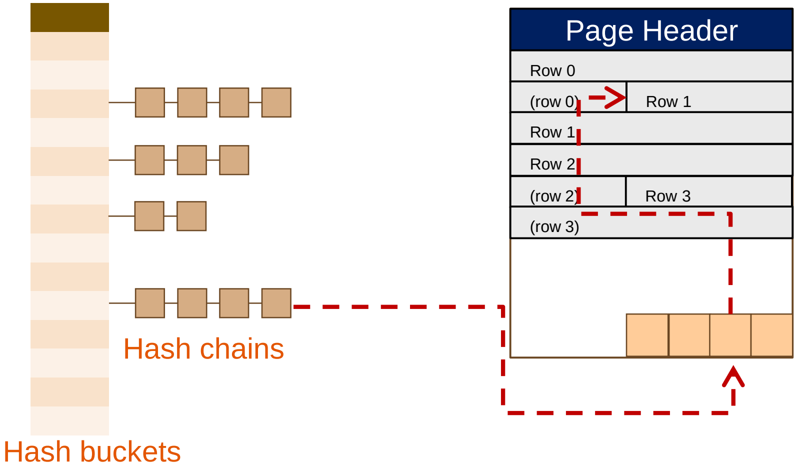
Hash

- Consists of hash table/buckets with hash chain of elements
- Hash nodes are composed of hashkey value + RID
  - ✓Note that the index key value is *not* in the hash table
- Used in ASE internals a lot (e.g. cache, locks)

Bitwise

- Consists of encoded bits packed into dense large IO pages
- Mainly used for analytics and especially for low cardinality values when expecting a large number of rows to qualify
- Not part of today’s discussion





B-Tree vs. Hash Index

B-Tree Index

Advantages

- Supports partial index usage
  - ✓ Or scans, etc.
- Supports range scans & min/max optimizations
  - ✓ Leaf nodes are sorted
- Supports non-unique values easier
  - ✓ Shorter RID list implementation
- Easier to maintain
  - ✓ As table grows, index tree can be pruned/expanded as necessary with impact minimized to local tree branches (rare root level impacts)

Disadvantages

- Slower - needs multiple string comparisons when traversing the b-tree
- Considerable space used - intermediate nodes have copies of index keys
  - ✓ This is reduced in ASE via suffix compression

Hash Index

Advantages

- Can be faster - hash operation & serial scan of hash chain
- Denser - hash values much smaller than index keys
  - ✓ Index keys only stored once on hash nodes

Disadvantages

- Doesn't work well on large tables as the number of hash buckets results in longer hash chains (and more time to scan)
  - ✓ Larger hash tables take up a lot of memory
  - ✓ Each hash chain value will need string comparisons (data page)
- Doesn't work well on non-unique indexes
- Doesn't support range, min/max, inequality predicates
  - ✓ Only works for equisargs and IN() lists
- Doesn't support partial index usage
  - ✓ Full index key values must be present for hash value derivation
- Harder to maintain
  - ✓ If hash buckets are increased, all index keys have to be re-hashed to determine new buckets

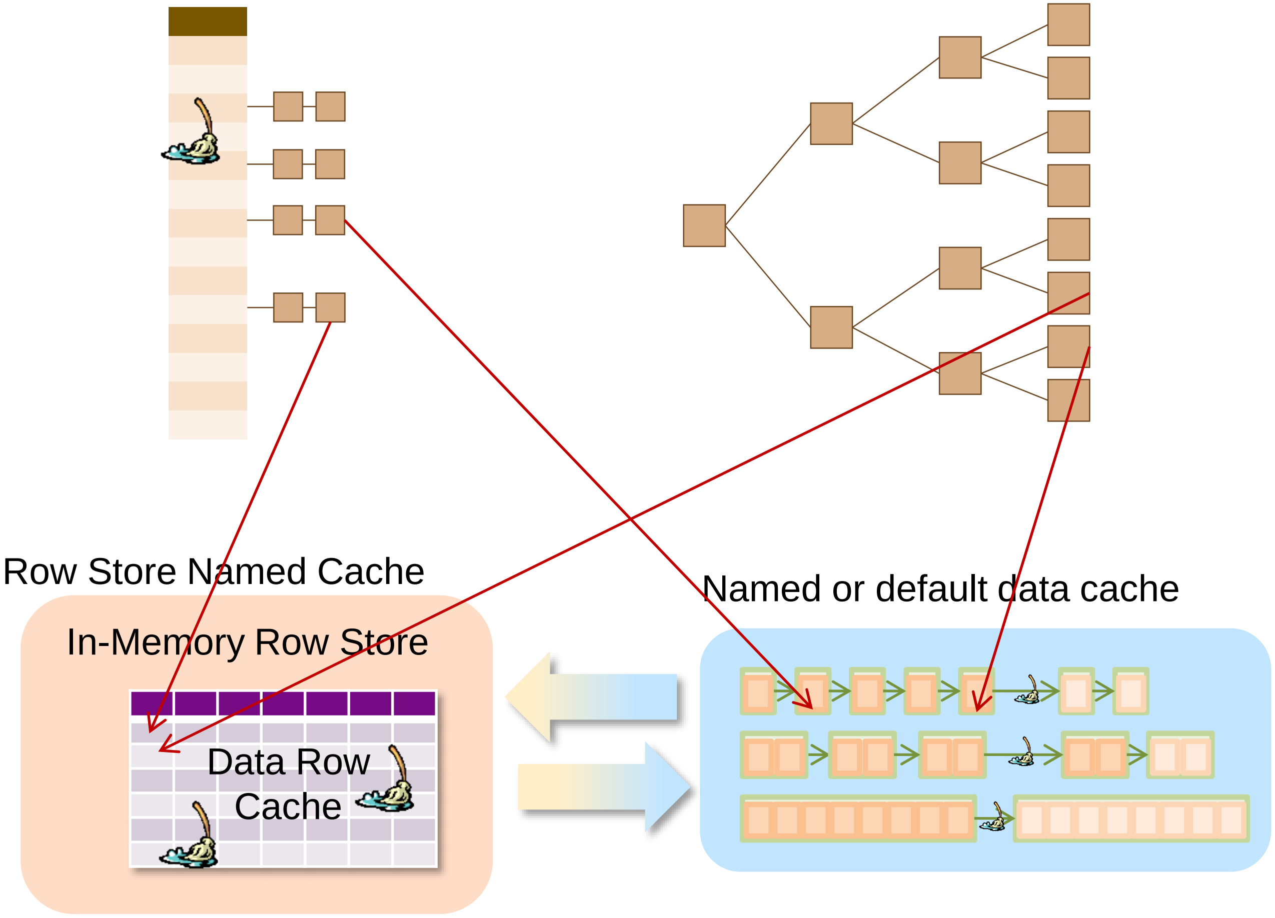
Best of Both Worlds: Hash Cache B-Tree (1)

Hash Cache B-Tree

- Only on unique indexes for IMRS enabled tables
- Rows promoted to IMRS will have unique index keys hashed and added to HCB
  - ✓Hash chain node points to IMRS row
- If row is demoted from IMRS, unique index keys stay hashed in HCB
  - ✓Hash chain node points to row in row offset table same as b-tree index page+RID

How it works

- If query uses equisargs or IN() & unique index
  - ✓Index keys are first checked if in HBC, if so, row in either IMRS or page cache is returned
  - ✓This includes JOINS!!!
- Otherwise, query uses b-tree index
  - ✓For non-unique indexes, this also could point to rows in the IMRS



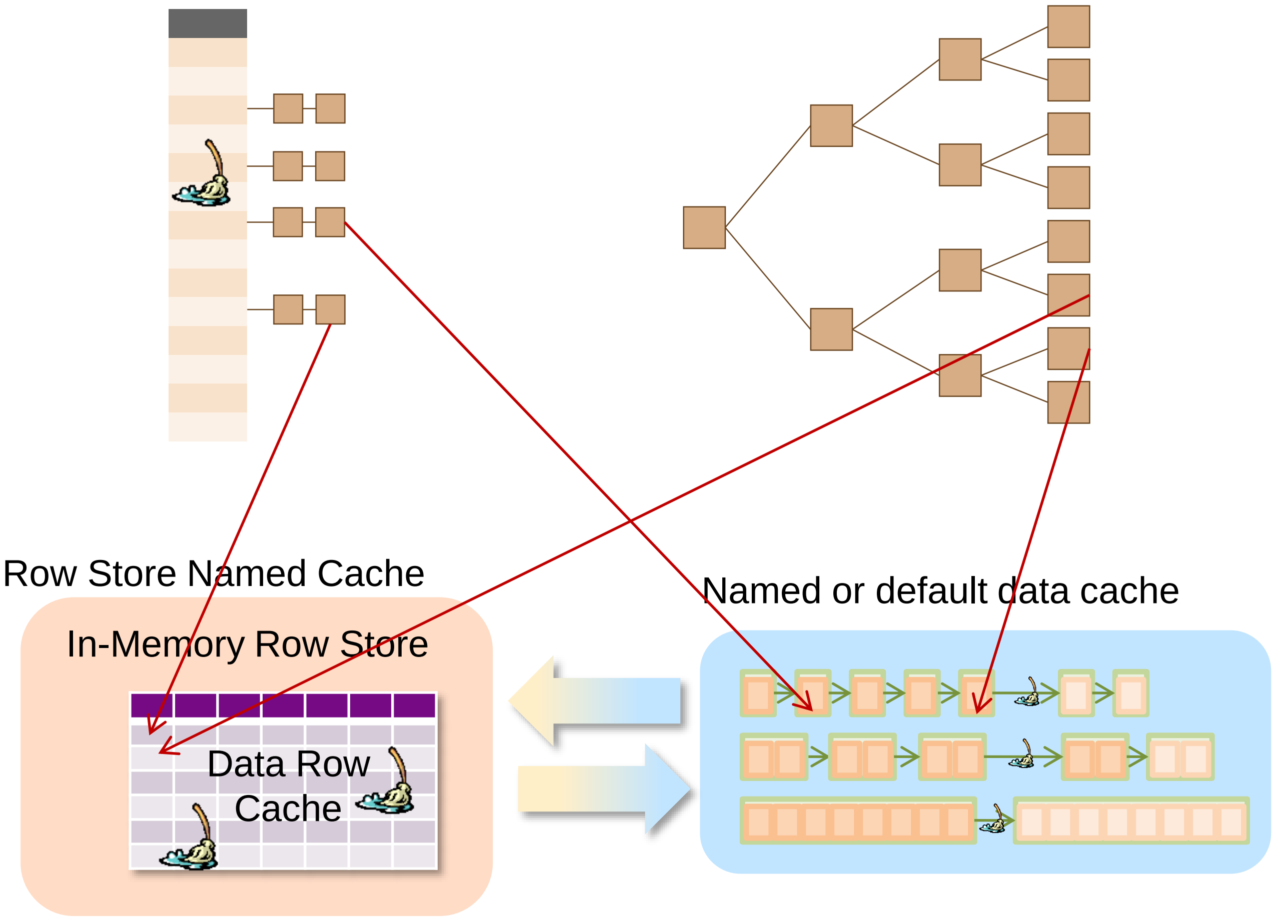
Best of Both Worlds: Hash Cache B-Tree (2)

Advantages over native Hash Indexing

- Since only IMRS cached rows are hash indexed, the number of hash buckets can be smaller yet still have short hash chains for fast access
- Advantages of B-tree index are retained eliminating need for duplicate indexes on key columns
  - ✓Think datetime cols in pkey index and range scans or min/max queries

Disadvantages

- Since cache volumes may fluctuate, more difficult to predict the number of hash buckets



Summary of differences between HCB & Native Hash Index

DML/Operation	B-Tree	HCB+B-Tree	Hash
All rows in index when created	Yes	B-Tree only	Yes
Recovered at server boot	Yes	B-Tree only	Yes
Works well with dynamic tables*	Yes	Yes	No
Works with really large tables*	Yes	Yes	No
Point query with all index keys supported	Yes	Yes	Yes
Point query with partial index keys supported	Yes	(uses B-Tree)	No
Range query supported	Yes	(uses B-Tree)	No

*This takes a bit of thinking. First, consider a table with 1B rows. If we only allow 1M hash buckets, each bucket would cover 1000 rows – far too many for any speed. Even if we reduce to 5 for hash chain length, it then takes 200M buckets which would take 40GB of memory and still not be very effective due to hash chain of 5. Reducing to ~2 would take 100GB of memory. It is possible (of course) to not cache the entire hash index, but this could result in physical reads on index access – which slows down OLTP. On the other hand, it is likely the active portion of the table would be less than 100K rows, which would only need ~4MB of memory for HCB with 1 bucket & 1 node per row

HCB Syntax

Only works on….

- HCB will only be used for tables (& rows) that are in the IMRS
- HCB will only work on tables that are IMRS enabled
- HCB will only work on unique indexes

HCB index settings

- ON → index will use HCB
- OFF → index will not use HCB
- DEFAULT → index may use HCB automatically via auto tuning

HCB bucket counts

- bucket_count is optional and always rounded up to power of 2 internally
- a value is calculated internally as below if it’s not specified
  - ✓ MAX(100000, (number_of_table_data_rows / number_of_index_partitions) * 1.5)

```
create unique [clustered | nonclustered] index index_name
 on [ [database.] owner.] table_name
             (column_expression [asc | desc]
             [, column_expression [asc | desc]] ...)
 [with { fillfactor = pct,
         latch_free_index = { ON | OFF },
         index_comression = { PAGE | NONE},
         index_hash_caching = { ON  [ , bucket_count = x  ]
              |  OFF | DEFAULT }
```

```
alter index [ [ database.] [owner.] table_name.index_name
set index_hash_caching = { ON  [ , bucket_count = x  ]
              |  OFF | DEFAULT }
```

HCB Operations

Which statements are supported

- Point query by
  - ✓SELECT
  - ✓UPDATE
  - ✓DELETE
  - ✓JOIN
- **All indexed columns** are specified using “=“ in the WHERE clause

HCB index rows

- Inserted rows – rows added to HCB due to IMRS inserts or point queries
- Deleted rows – rows removed by deletes, reorgs or other maintenance operation

HCB Garbage Collection

- Frees up memory from deleted HCB rows
- By default, there are 2 tasks (ASE spids) per database

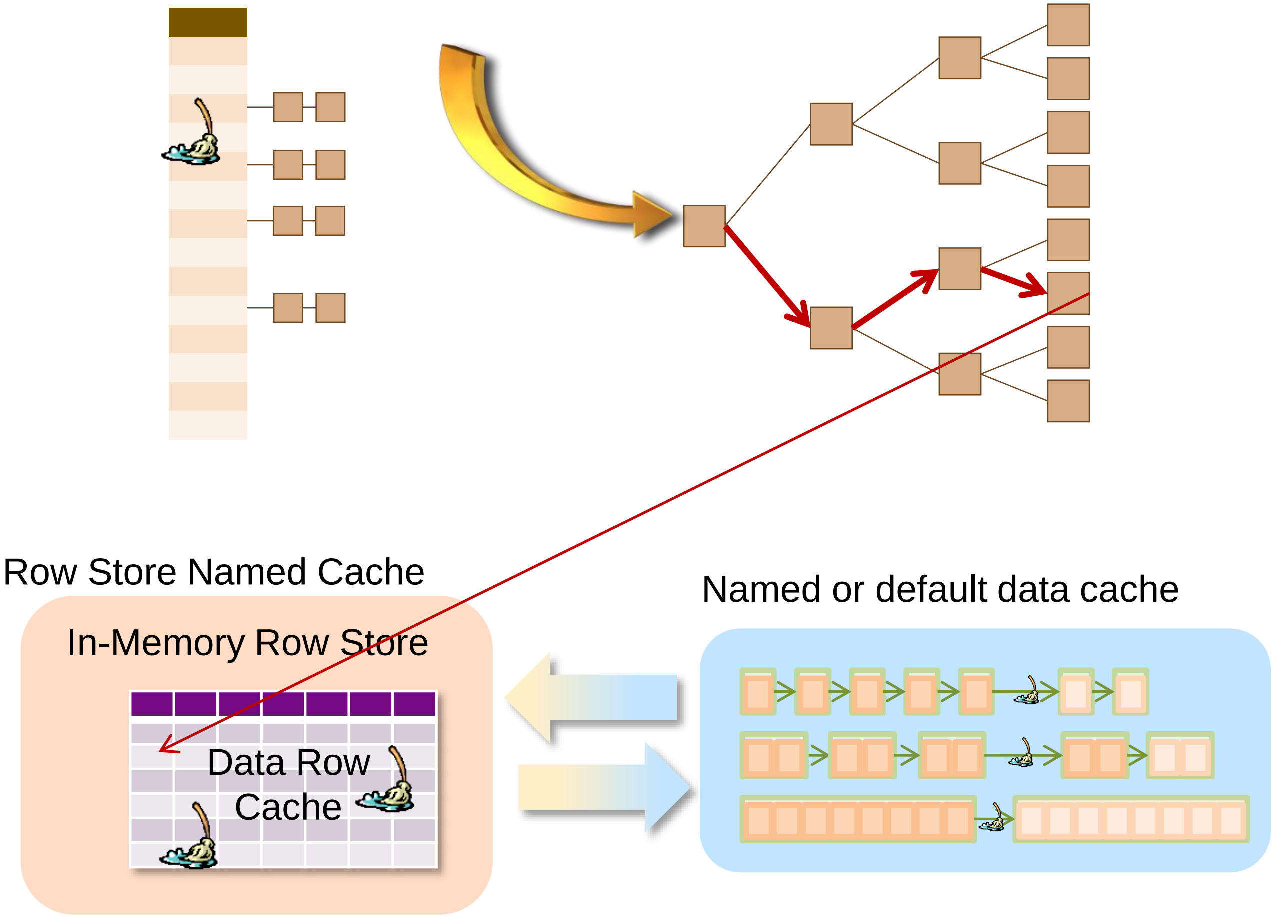
HCB & SQL operations: Insert

Initialization

- Empty hash table with a fixed number of buckets is initialized when first point query comes in

DML Insert

- **Row is inserted into IMRS**
- **Index row is inserted into B-Tree**
  - ✓Go through BTree index as usual
    - we don't have a RID yet, so can't put in HCB
    - Remember, clustered index may not be the unique index….and there may be more than one unique index
  - ✓Index keys migrated to HCB on first point query after select



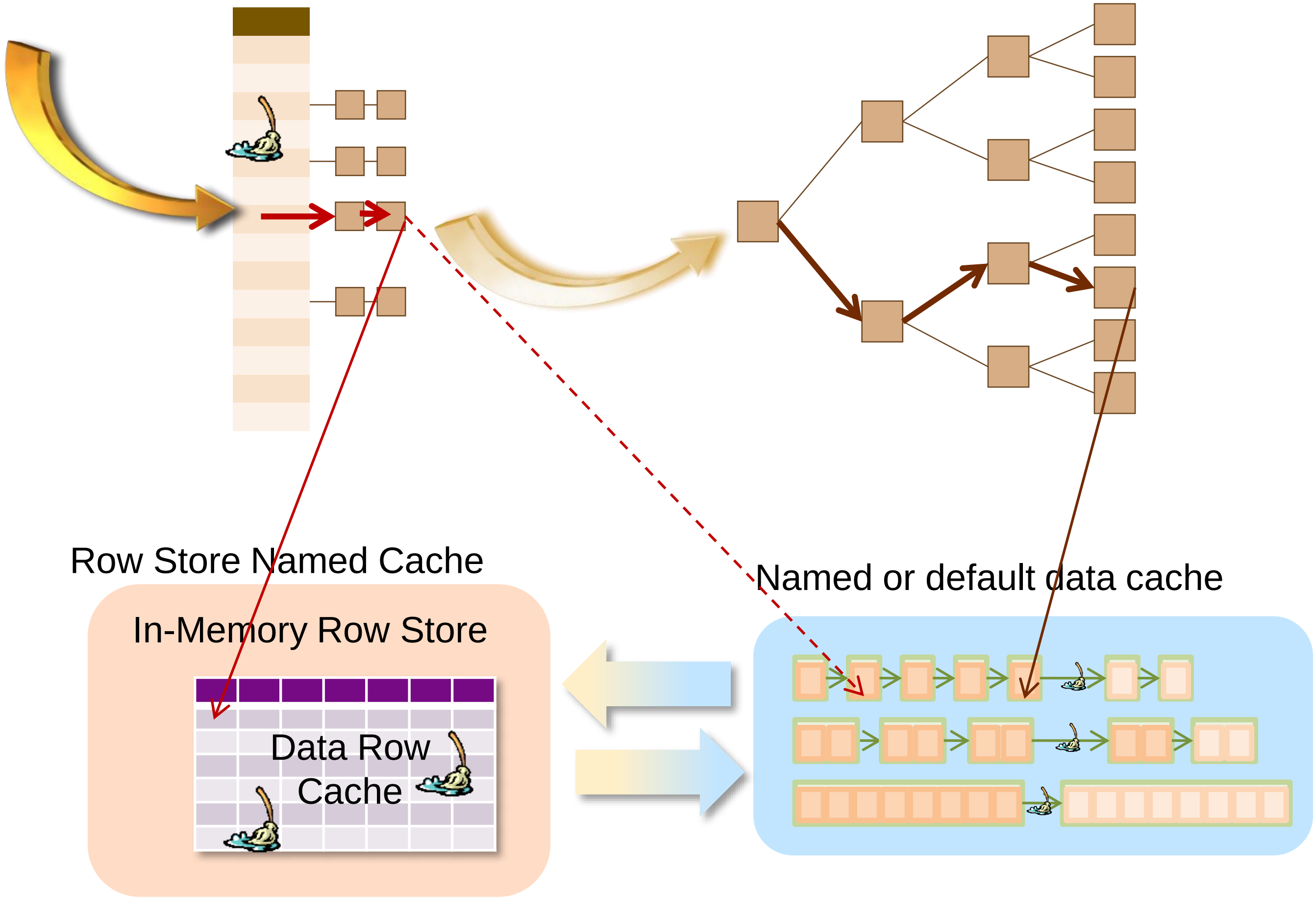
HCB & SQL operations: Select

If point query

- Firstly search hash index cache
- If found qualified node in hash index cache, check in IMRS or data pages to see if data row matches
  - ✓Once an IMRS row is added to HCB, the fact that the row may have been repacked to page cache does not remove HCB entry – only HCB GC does
- Else, search BTree index and go through BTree index regular process.
- If finally a qualified index row is found in BTree index, cache the index row into hash index cache

If not point query

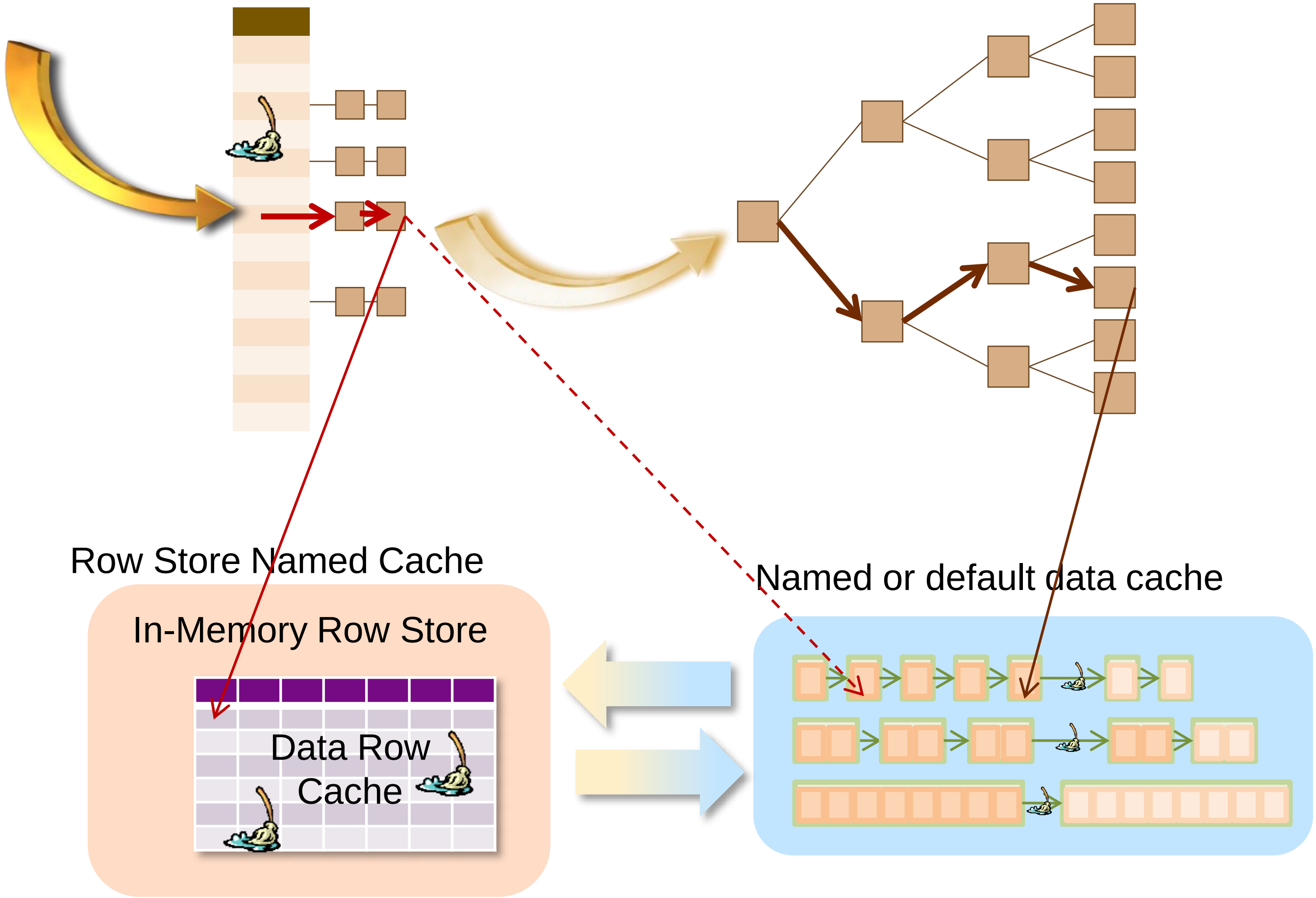
- Search Btree index, but do not cache index rows in HCB



HCB & SQL operations: Update

Update

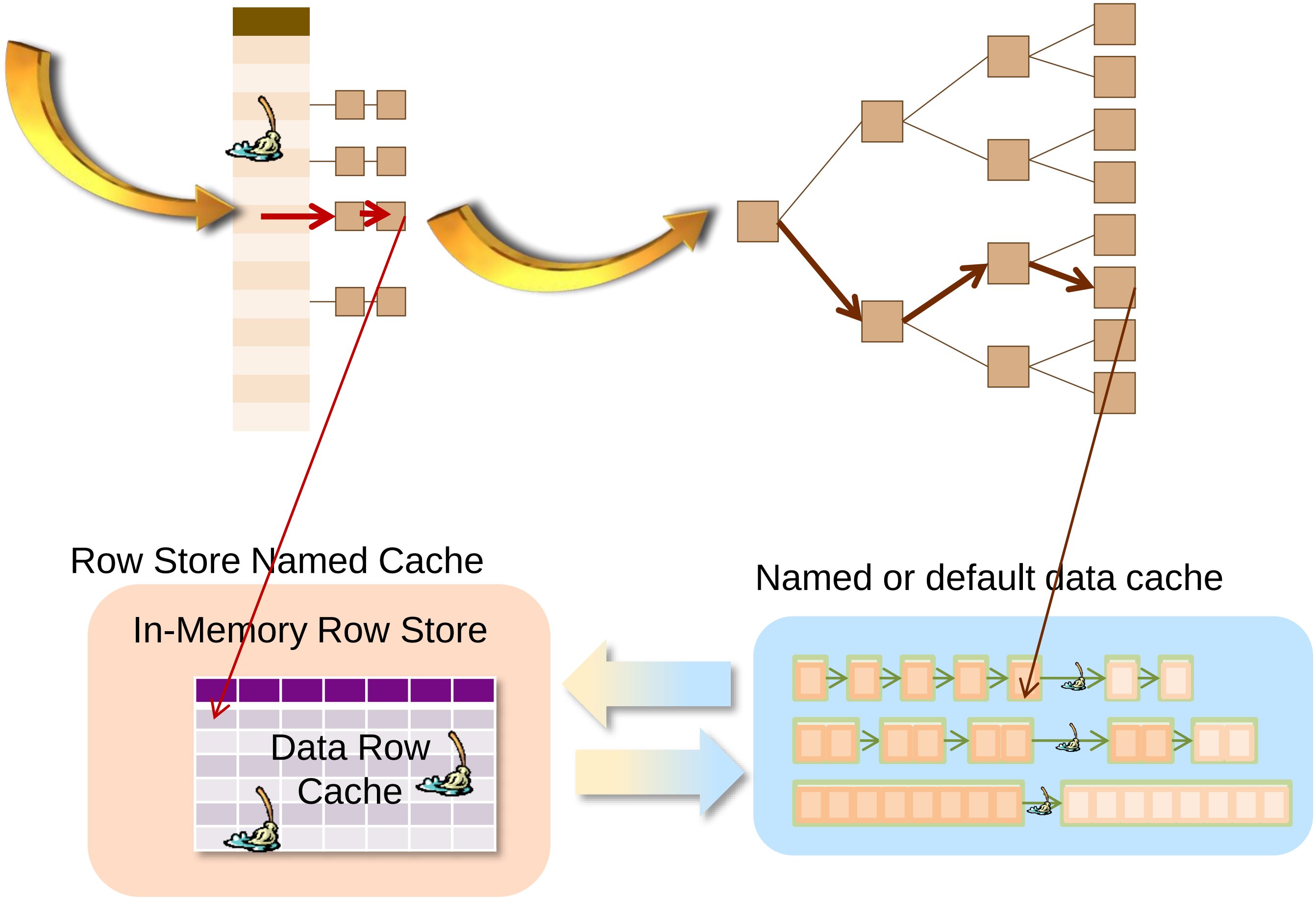
- Firstly search hash index cache if point query
- If found qualified node in hash index cache, check in IMRS or data pages to see if data row matches, and go through regular update process
- Else, search BTree index and go through regular update process.
- If finally a qualified index row is found in BTree index and if the update is non-key update, cache the index row into hash index cache
- Special cases
  - ✓ If indexed columns are updated, delete corresponding hash index node
  - ✓ If data row jumps crossing partitions, delete corresponding hash index node
  - ✓ If row forwarding, do nothing



HCB & SQL operations: Delete

Delete

- Delete hash node firstly
- Then delete BTree index entry as usual



Configuring HCB

IMRS must be enabled

- HCB will only be used for tables (& rows) that are in the IMRS

HCB is not stored in IMRS

- Uses a memory pool in ASE
  - ✓Via the memory buckets you see in the config file
- Default is 8MB
- Sizing aspects
  - ✓Hash bucket → 8 bytes
  - ✓Hash node → 40 bytes

Supports a notion of autotuning

- Turns on/off certain operations per table based on workload
- For example, it may disable for inserts if rows inserted into IMRS are not being re-read

```
-- enabling
sp_configure ‘enable HCB index’, 1
sp_configure ‘HCB index memory pool size’, <new value>”

-- tuning (other than sizing)
sp_configure ‘number of hcb gc tasks per db’, <new value>

-- auto tuning
sp_configure ‘HCB index auto tuning’, 1
sp_configure ‘HCB index tuning interval’, <new value>
```

Monitoring & Sizing

MDA Tables

- monHCBPartitionActivity
- monHCBGCTasks

Low-level bucket pools

- Either use the bucketpool() system function
- ….or the MDA table monBucketPool

Sizing

- Number of buckets * 8 = hash table memory
- Number of nodes * 40 = hash node memory
- Total (for any index) = (number of buckets * 8) + (number of nodes * 40)
- Estimate = Number_of_configed_buckets * 8 + (Number_of_data_rows * percentage_of_hot_rows) * 40
- Increase if HCB memory pool is >80% used

```
-- Get the total size of bucket pool
select bucketpool(”HCB Index Memory Pool/size")

-- Get the current used size of bucket pool
select bucketpool("HCB Index Memory Pool/used")

-- Get the max used size of bucket pool since ASE server is booted
select bucketpool("HCB Index Memory Pool/used_max")

-- Get the available memory size of the bucket pool
select bucketpool("HCB Index Memory Pool/bucket/0/available")

-- monBucketPool (for sizing)
select  NumBuckets, NumInstances, BucketPoolSize,
          BucketPoolUsed, BucketPoolUsedMax,
          BucketPoolOverhead, BucketSize
from master..monBucketPool
where BucketPoolName like "%HCB Index%”
```

MVCC

Multi-Version Concurrency Control

MVCC Details

Support is limited to….

- Datarows locked tables only
- Datarows locked tables in user databases
  - ✓ Not system tables and not tempdb tables

Will support both IMRS and traditional disk/page-based tables

- IMRS → row versions are stored in database linked row store
- Traditional disk/page-based → row versions are stored in linked tempdb

Typical ANSI Isolation Levels (part of SQL standard):

- “READ UNCOMMITTED” (Level 0)
- “READ COMMITTED” (Level 1)
- “REPEATABLE READS” (Level 2)
- “READ SERIALIZABLE” (Level 3)

Typical MVCC Isolation Levels (not part of SQL Standard)

- “STATEMENT SNAPSHOT” → ISO level 1
- “TRANSACTION SNAPSHOT” → ISO level 2
- “READONLY STATEMENT SNAPSHOT” → ISO level 1

IMRS + DRC/MVCC

A table in IMRS can be DRC, MVCC or both

- DRC - data row caching (“hot data”)
- MVCC - versioning in IMRS

Syntax

```
create table
[[database.][owner.]]tablename
     (column_name datatype
      [, column_specification ...]
     )
lock datarows
[with [row_caching { ON [ FOR { ALL | SOME
} ROWS ] | OFF } ]
[, snapshot_isolation { ON | OFF }
 [ USING row storage] ]
```

SQL	DRC	MVCC	Both
Inserts	IMRS	IMRS	IMRS
Freq queried rows	IMRS	Page	IMRS
Low updated rows	Page	IMRS	IMRS
Freq updated rows	IMRS	IMRS	IMRS
Deletes	Page	IMRS	IMRS

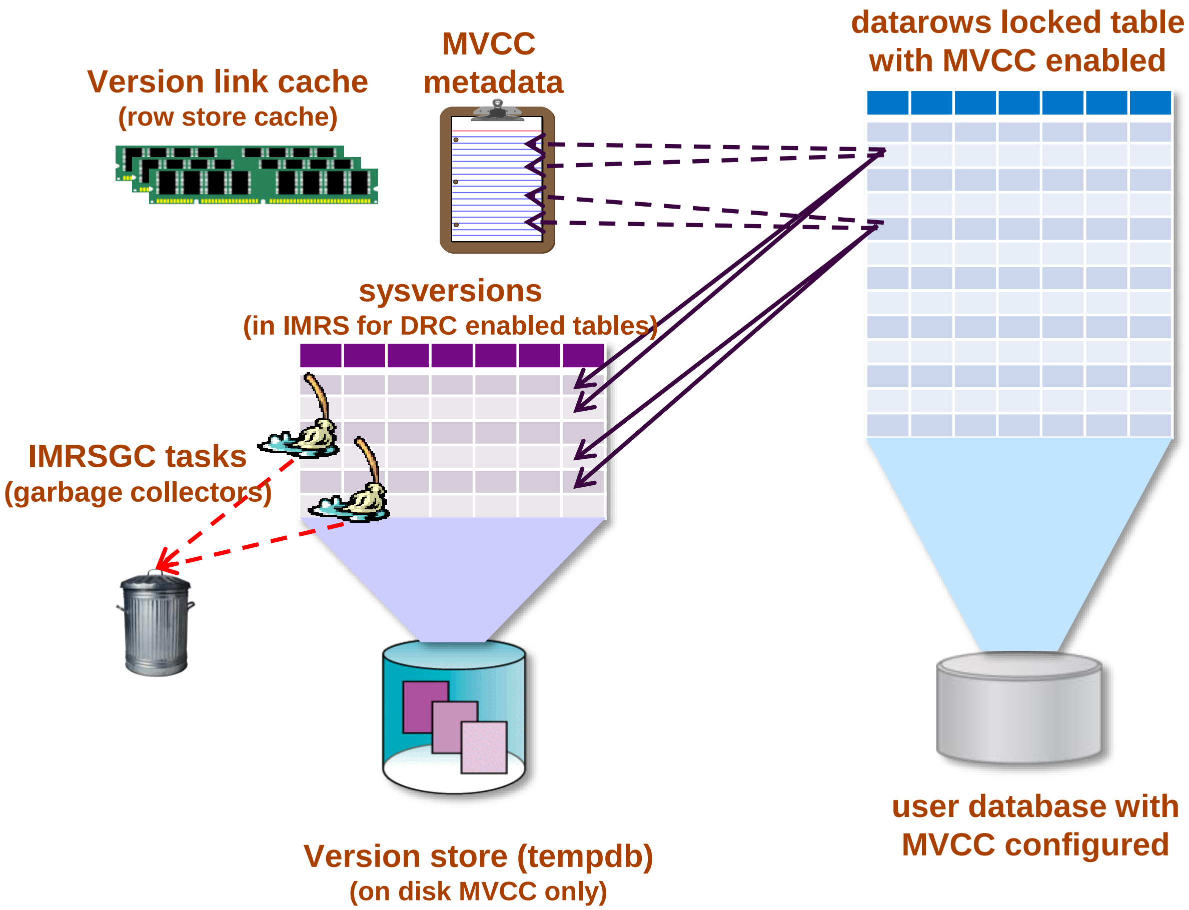
Implementing MVCC

At a high level

- We have a version link cache which contains the MVCC meta data that links the versions and tracks the version status
- Version store – this can be the IMRS store itself or a dedicated tempdb depending on table's DRC status
- IMRSGC tasks – responsible for clearing old versions – whether table is DRC or on-disk

Database has to be altered to support MVCC

- Needs a row store cache for MVCC metadata
  - ✓ Aka "version link cache"
  - ✓ master..sysdatabases.imrscache column
- DRC + MVCC enabled tables use IMRS cache for version store – no separate tempdb needed
- On-disk MVCC enabled tables need a separate tempdb for row versions
  - ✓ 1:1 mapping per database with MVCC enabled
  - ✓ Current row version is on page
  - ✓ Old version is moved to sysversions table in version store



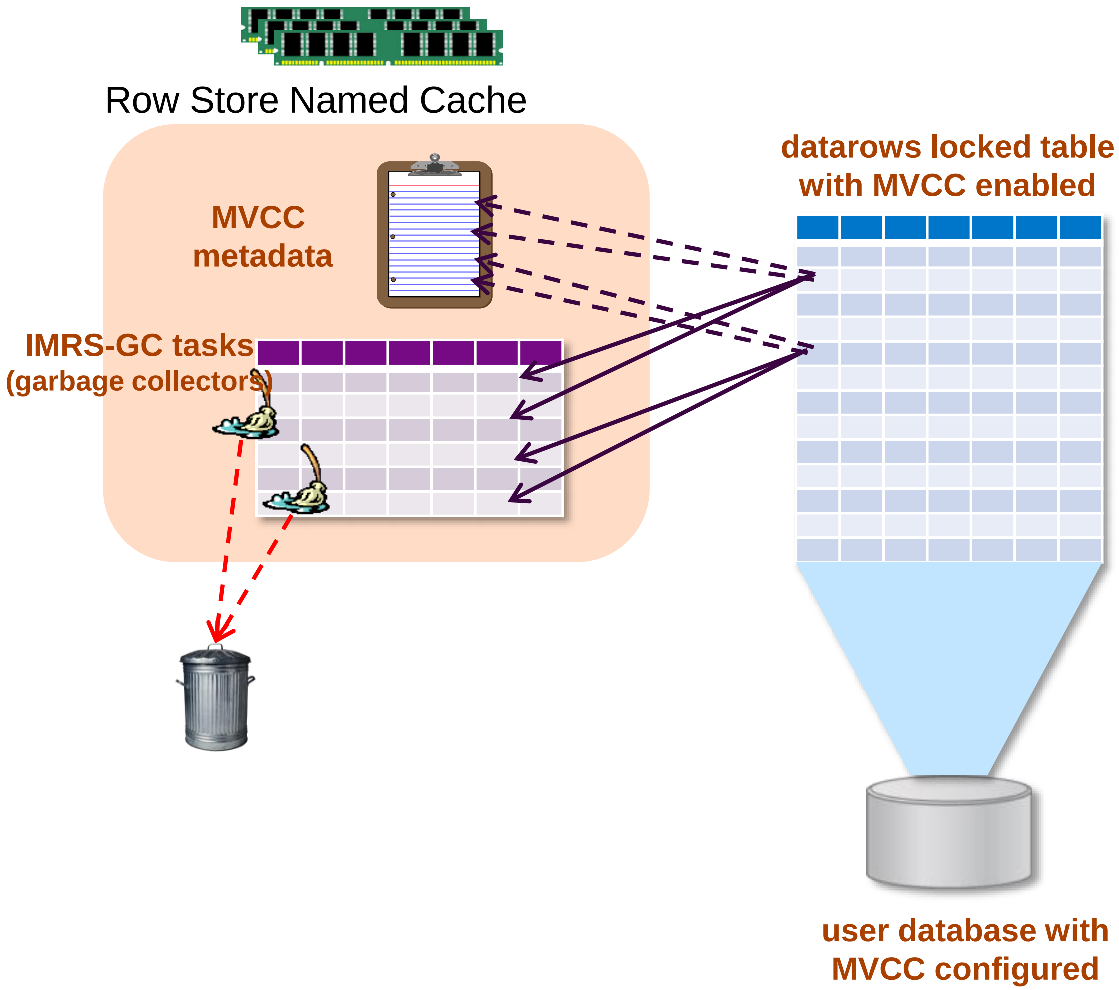
Implementing MVCC for DRC Tables

Row versions are kept in IMRS cache

- GC tasks will erase
- You will need a larger IMRS cache than just for DRC since versions will consume memory

IMRS Cache holds the MVCC metadata

- This still can be considerable memory (GB’s)


Row Store Named Cache
MVCC metadata
IMRS-GC tasks
(garbage collectors)
datarows locked table with MVCC enabled
user database with MVCC configured

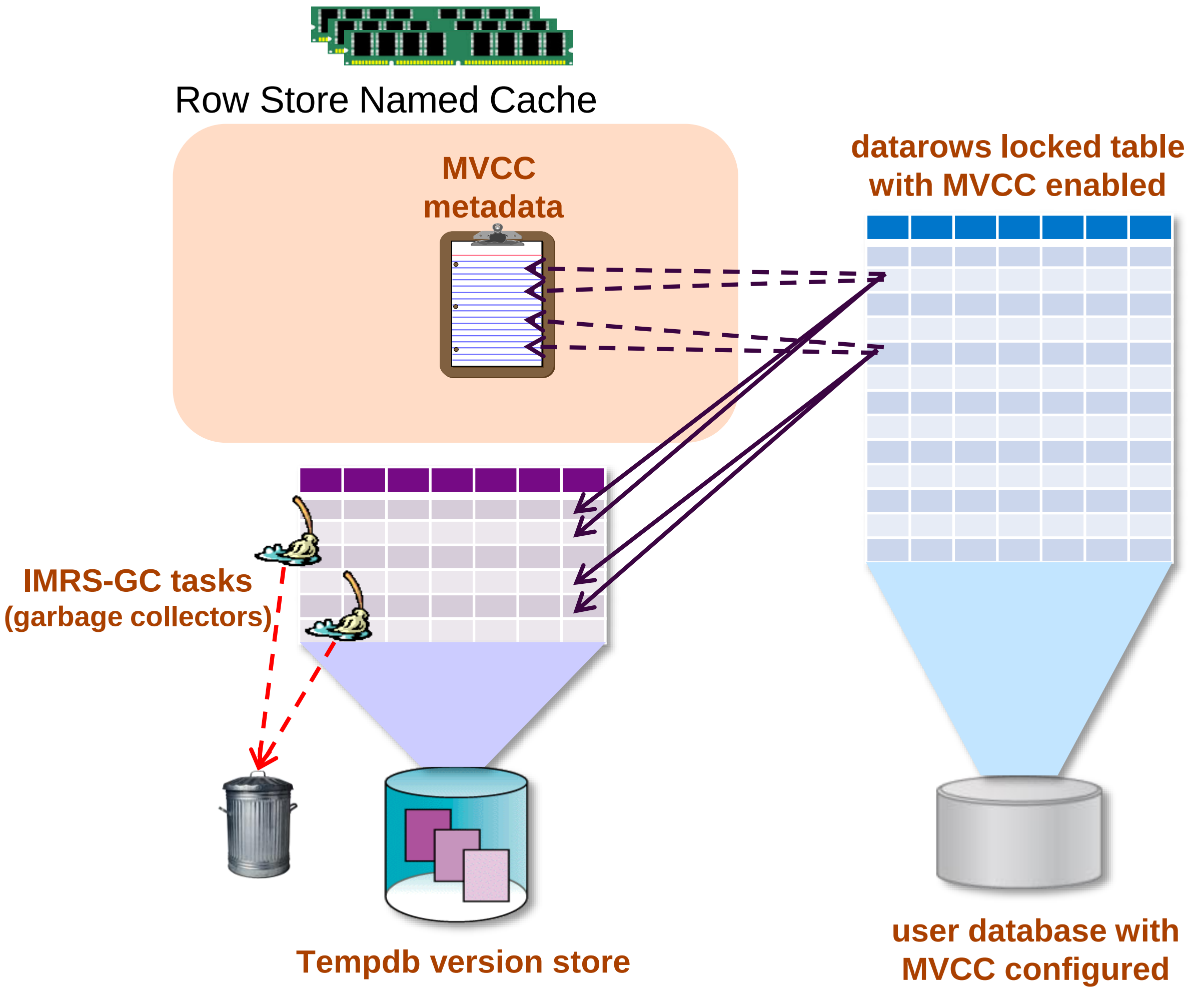
Implementing MVCC for on disk (page-based) tables

Row versions are kept in tempdb version store

- GC tasks will erase
- Only old versions are in tempdb version store
  - ✓Current versions are in actual page
  - ✓Makes recovery easier (version store cleared on reboot)

IMRS Cache holds the MVCC metadata

- This still can be considerable memory (GB's)
- Think about it – metadata would include
  - ✓DBID – 2-4 bytes
  - ✓Object ID – 4 bytes
  - ✓Page ID – 4 bytes
  - ✓Row ID – 2-4 bytes
  - ✓Transaction ID – 32 bytes
  - ✓Version # - 2-4 bytes
  - ✓Version Status – 2+ bytes



Row Store Named Cache
MVCC metadata
datarows locked table with MVCC enabled
IMRS-GC tasks (garbage collectors)
Tempdb version store
user database with MVCC configured

Defining the version stores

Version link cache (row store)

- Minimum size is 128*pagesize
- As with all caches
  - ✓Size can be increased dynamically
  - ✓Decreasing size requires a reboot (static)

Version store (tempdb)

- Size depends on size of tables * number of concurrent versions needed

Use create database/alter database to enable

- Specify the version link cache
- Specify the version store

```
sp_cacheconfig [ cache_name ]
    [, “ cache_size [ P | K | M | G ]” ]
    [, logonly | mixed | inmemory_storage | row_storage ]

sp_cacheconfig "pubs2_vlink_cache", "20g", "row_storage"

-- Create a temporary database for version store
create temporary database pubs2_version_store
      on mycached_fs_device=‘100G’

-- create the database/configured for MVCC
create database production
        on data_dev1 = …
        log on log_dev1 = …
row storage on pubs2_vlink_cache
version storage on pubs2_version_store
-- optionally if we add this clause then all new datarows
-- locked tables will have MVCC enabled by default
with snapshot isolation on
```

Syntax and Levels

Database level

- Defines row cache for versions of IMRS tables
- Defines tempdb for versions of disk/page tables
- Existing databases can be altered but need to be in single user mode

Table level

- Can only be enabled if enabled at database level

Session level

- Only affects tables with snapshot isolation enabled.
- Primarily to specify which snapshot isolation you want to use

```
CREATE TABLE [[database.][owner.]]tablename
(column list> ...)
WITH SNAPSHOT_ISOLATION ON
    USING VERSION STORAGE

alter table [[database.][owner.]]tablename
set snapshot_isolation { ON
    [USING {ROW | VERSION} STORAGE] | OFF }

SET TRANSACTION ISOLATION LEVEL {
    STATEMENT SNAPSHOT | TRANSACTION SNAPSHOT |
    READONLY STATEMENT SNAPSHOT |
    READ UNCOMMITTED | READ COMMITTED |
    REPEATABLE READ |
    SERIALIZABLE | 0 | 1 | 2 | 3}
```

Row Version Commit Timestamp

Each versioned row will have timestamp

- Uncommitted = infinity
- Committed = commit timestamp

Snapshots

- Query only sees data with timestamp <= snapshot time stamp

Snapshot Isolation

- Statement Snapshot & Readonly Statement Snapshot → begin timestamp
- Transaction Snapshot → timestamp of first DML or scan/query in transaction
  - ✓ Think of it like chained mode - the first statement that starts a transaction defines the timestamp used for snapshot isolation

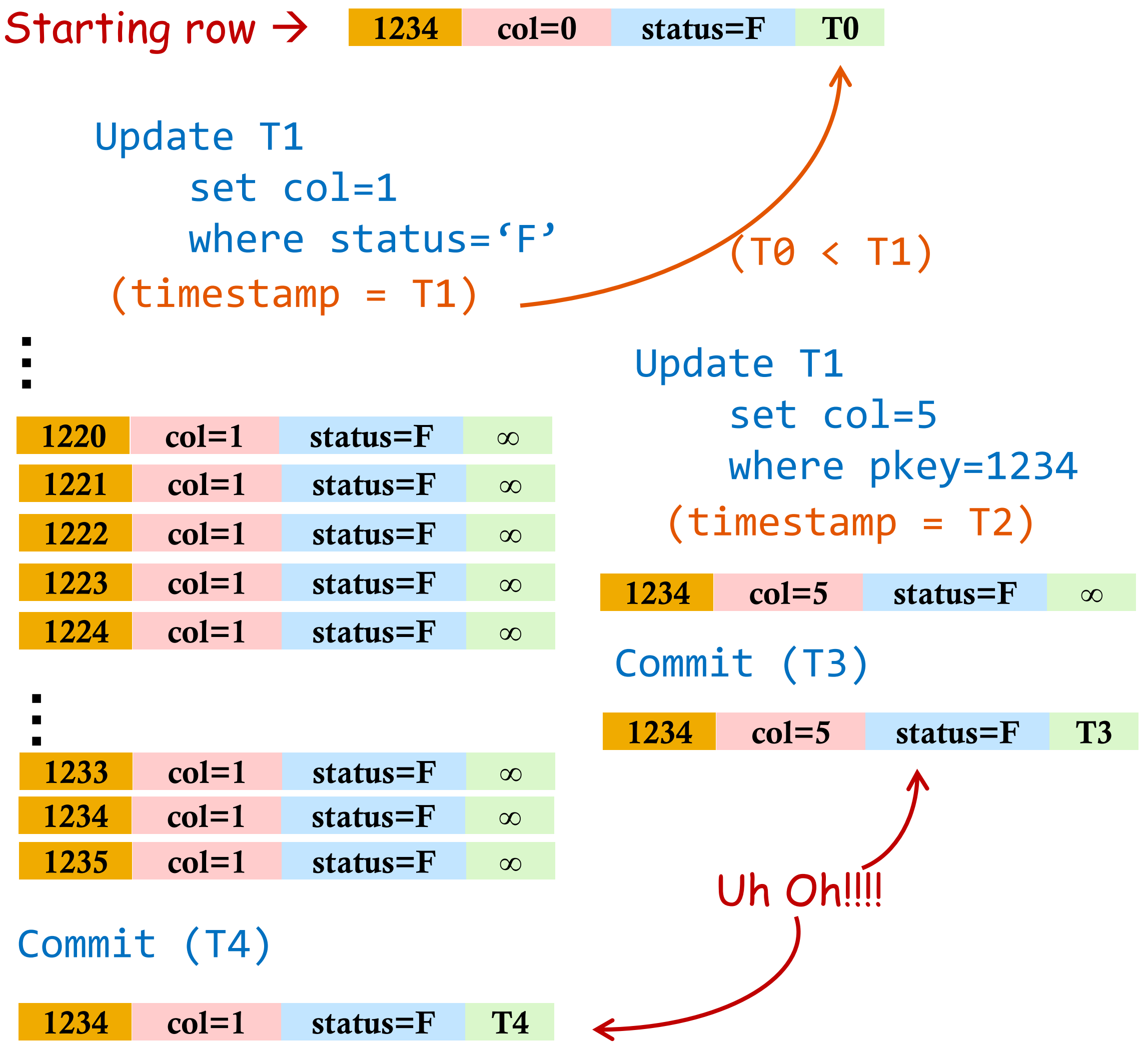
Optimistic vs. Pessimistic MVCC

Optimistic

- Writers take no locks
- Disadvantage - if multiple users change the same row, there could be version inconsistencies and both would need to be rolled back
- For in-memory tables, MS uses optimistic

Pessimistic

- Writers use page/row locks as usual
- Writers don't block readers due to versioning
- Rolls back if later version of row exists
  - ✓Theoretically, the problem isn't so much on commits as it is rollbacks
  - ✓For example, what happens if we rollback at T4
    - – Do we rollback the row to T0 image???
- For write conflicts we try to mimic Oracle behavior



Starting row →

1234	col=0	status=F	T0

```
Update T1
    set col=1
    where status='F'
(timestamp = T1)
```

(T0 < T1)

⋮

1220	col=1	status=F	∞
1221	col=1	status=F	∞
1222	col=1	status=F	∞
1223	col=1	status=F	∞
1224	col=1	status=F	∞

⋮

1233	col=1	status=F	∞
1234	col=1	status=F	∞
1235	col=1	status=F	∞

Commit (T4)

1234	col=1	status=F	T4

```
Update T1
    set col=5
    where pkey=1234
(timestamp = T2)
```

1234	col=5	status=F	∞

Commit (T3)

1234	col=5	status=F	T3

Uh Oh!!!!

Pessimistic Locking in ASE

Update with Statement Snapshot

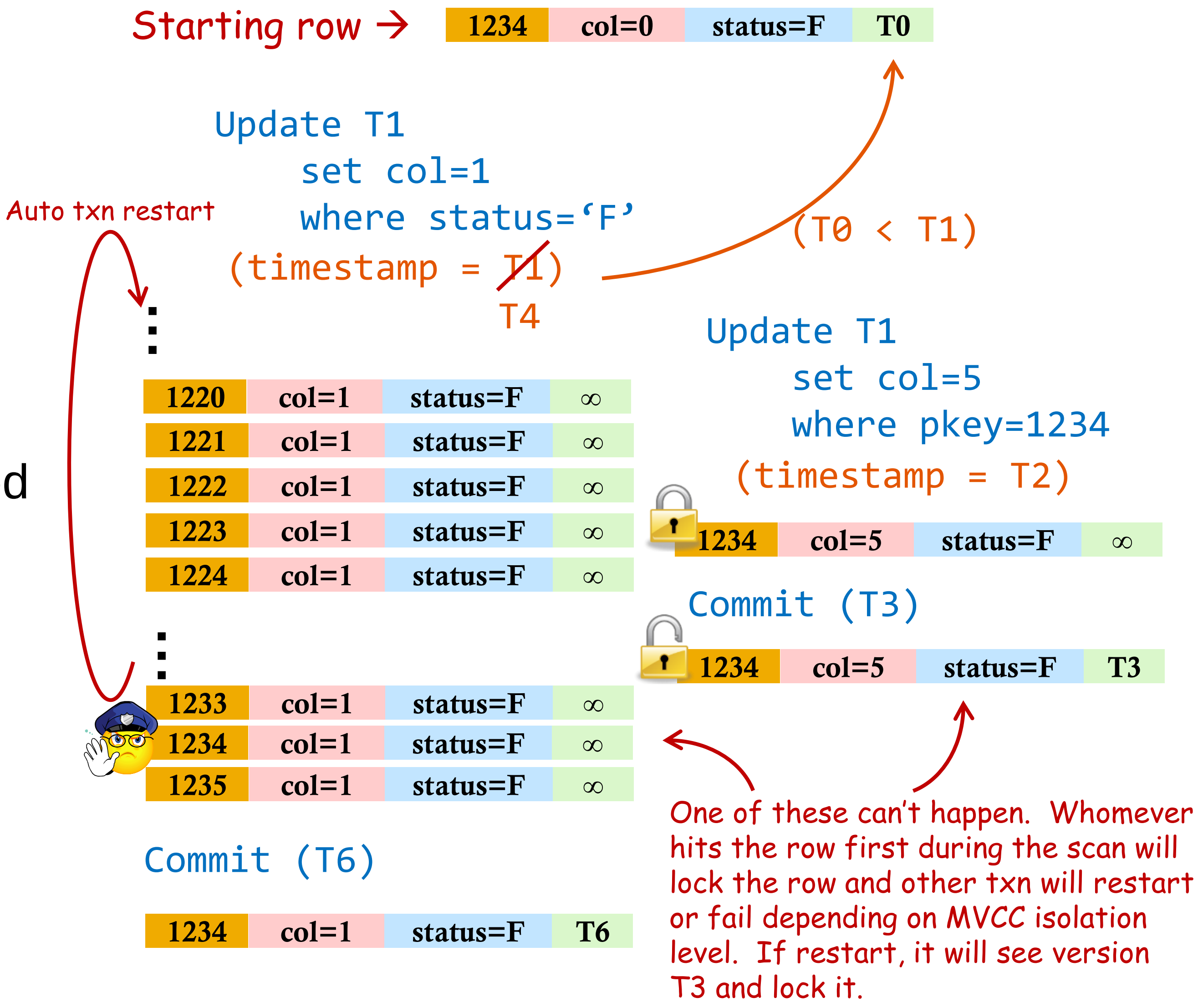
- Locks are acquired during scan phase
  - ✓If row TS is > statement TS, statement uses new timestamp and restarts
- Locks are held during any restarts

Transaction Snapshot

- In case of write conflict, only statement is rolled back - not transaction
- App gets ‘non serializable’ error
- App/user can rollback or retry

Readonly Statement Snapshot

- No retry as it uses latest committed version
- Rows are locked on scan immediately



Starting row →

1234	col=0	status=F	T0

Update T1
    set col=1
    where status=‘F’
(timestamp = ~~T1~~)
T4

(T0 < T1)

Auto txn restart

⋮

1220	col=1	status=F	∞
1221	col=1	status=F	∞
1222	col=1	status=F	∞
1223	col=1	status=F	∞
1224	col=1	status=F	∞

⋮

1233	col=1	status=F	∞
1234	col=1	status=F	∞
1235	col=1	status=F	∞

Commit (T6)

1234	col=1	status=F	T6

Update T1
    set col=5
    where pkey=1234
(timestamp = T2)

1234	col=5	status=F	∞

Commit (T3)

1234	col=5	status=F	T3

One of these can’t happen. Whomever hits the row first during the scan will lock the row and other txn will restart or fail depending on MVCC isolation level. If restart, it will see version T3 and lock it.

Version Garbage Collection

Multiple versions of the same row consume memory/space

- each insert / update / delete for tables with snapshot isolation results in creating a new version of data row in version store.
- Since each version has a timestamp associated with it and visibility of versions depends on version timestamp as well as scan timestamp, memory occupied by version links and disk space occupied in version tempdb by these versions can't be freed immediately.
- Server maintains a queue of scans and transaction snapshot transactions sorted on their begin timestamps.
  - ✓ The timestamp of the oldest element in this queue determines which data row versions can be garbage collected, since any data row versions which have timestamp more recent than this can be potentially 'visible' to any of the scans or transactions queued in this list.

Summary

Early results, etc.

When to use which XOLTP feature?

Feature	Useful Workload Scenarios
LLDC	Most concurrent workloads where relaxed cache can be used effectively
DRC	High Concurrent Insert Load High Concurrent Updates Small/lookup Tables Queue Tables Compressed Tables
HCB	Point query lookups using unique index – common in almost any OLTP workload Work/temp table joins to main table using pkey for join
LFB	Concurrent Index Scans (Almost all concurrent workloads)
SNAP	Scalar Aggregate, Nested Loop Joins, Large SELECT List, Loops in stored procedures
MVCC	Reporting applications Snapshot Application Porting/Compatibility Applications with locking problems

E2E Performance with 64 engines & minimal tunings*

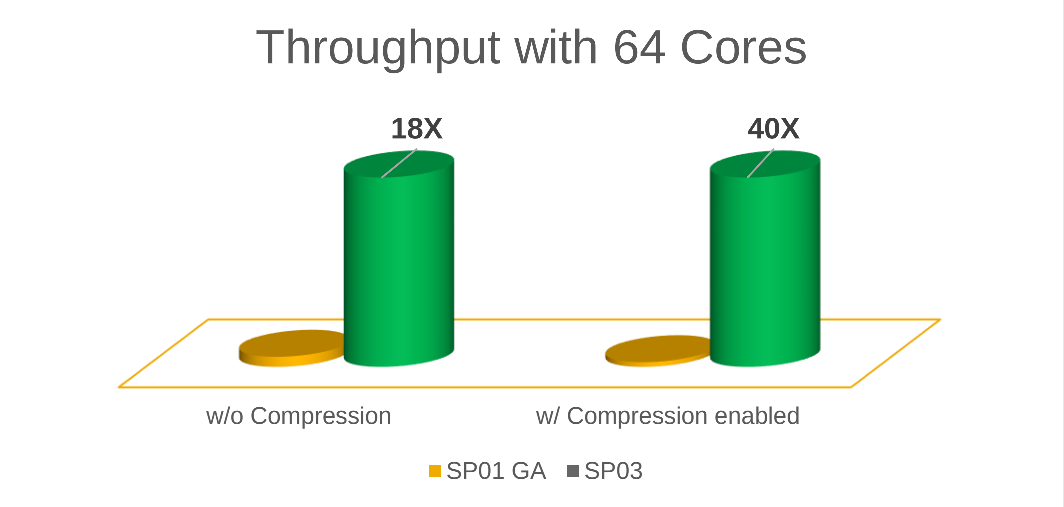
Common Industry Benchmark

ASE version	New order transactions / min	CPU Utilization	Comments
15.7 SP64	42K	99%	Cache manager spinlock contention
16.0 SP01 GA	42K	99%	Same as above
16.0 SP02 GA (with LFB+LLDC+SNAP)	282K (6.5X vs. baseline)	45%	Buffer unpinning on data pages, leading to latch conflicts resulting in syslogs semaphore contention
16.0 SP03 with same features	315K (7.5X vs. baseline)	44%	LFB Improvements; data page latch contention remains
+ DRC	719K (17X vs. baseline)	95%	Datapage latch contention removed
+ HCB	771K (18X vs. baseline)	96%	Further codepath improvements with HCB

That's ~13K tps….

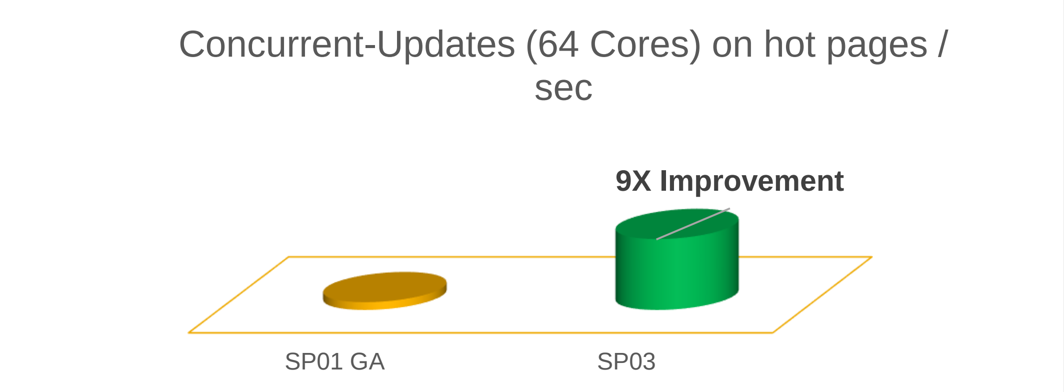
*Minimal tunings – in the past when running this benchmark, the system was aggressively tuned using schema techniques such as partitioning, max rows per page etc. to try to avoid contention. For these tests, only standard memory/proc cache and server configurations were tuned.

Record E-2-E Throughput Gains



- **Out of the box scaling Improvements**
  - ✓ SP02 scaling improvements:
    - Latch Free Btree (LFB)
    - Lockless Data Cache (LLDC)
  - ✓Further scaling improvements SP03:
    - In-Memory Row Cache
    - Hash Indexing
    - Compression code-path improvements

Lab results with TPCC style workload



- **Highly concurrent hot data access**
  - ✓ Scaling improvement by avoiding page level latch conflicts using ‘In-Memory Row Cache’

ASE XOLTP vs. MS SQL

XOLTP Feature	ASE 16 sp03	MS SQL 2014/2016
Compiled queries	Fully prepared stmts Stmt Cache Queries in Stored Procs	Stored procedures only
Compile query support	Any table/view/proc but limited QP operators	Only in-memory tables, limited support of proc language
In-Memory Row Store	Dynamic & tiered from any table	In-memory tables only
Replication of In-Memory Row Store	Supported	Not supported nor Always-On
Indexing on In-Memory Row Store	Latch-free B-tree on any table	Primarily hash with modified bw-tree
Index & cache locking optimizations	Any table/index; any cache	In-memory tables only (MVCC)
Concurrency on In-Memory Row Store	MVCC in tempdb with metadata in IMRS cache	MVCC in tempdb with in-row meta data/links to versions (in table overhead)
SSD Buffer Cache Extension	Read/Write w/ MACC	Clean pages only
Application Compatibility (e.g. SAP)	Full backward compatibility	App rewrite needed

For more information on SAP ASE 16 visit:

www.sap.com/ase

http://help.sap.com/ase1602/

https://ideas.sap.com/SAPASE

Jeff Tallman
jeff.tallman@sap.com