Db2 Architecture. Overview and BLU

Keri Romanufa

IBM

Session code: TRIDEX

06/13/2019 2:30

Db2

IDUG

Leading the Db2 User Community since 1988

Please note:

- IBM’s statements regarding its plans, directions, and intent are subject to change or withdrawal without notice and at IBM’s sole discretion.
- Information regarding potential future products is intended to outline our general product direction and it should not be relied on in making a purchasing decision.
- The information mentioned regarding potential future products is not a commitment, promise, or legal obligation to deliver any material, code or functionality. Information about potential future products may not be incorporated into any contract.
- The development, release, and timing of any future features or functionality described for our products remains at our sole discretion.
- Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon many factors, including considerations such as the amount of multiprogramming in the user’s job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve results similar to those stated here.

Notices and disclaimers

- © 2019 International Business Machines Corporation. No part of this document may be reproduced or transmitted in any form without written permission from IBM.

- **U.S. Government Users Restricted Rights — use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM.**

- Information in these presentations (including information relating to products that have not yet been announced by IBM) has been reviewed for accuracy as of the date of initial publication and could include unintentional technical or typographical errors. IBM shall have no responsibility to update this information. **This document is distributed "as is" without any warranty, either express or implied. In no event, shall IBM be liable for any damage arising from the use of this information, including but not limited to, loss of data, business interruption, loss of profit or loss of opportunity.** IBM products and services are warranted per the terms and conditions of the agreements under which they are provided.

- IBM products are manufactured from new parts or new and used parts. In some cases, a product may not be new and may have been previously installed. Regardless, our warranty terms apply."

- **Any statements regarding IBM's future direction, intent or product plans are subject to change or withdrawal without notice.**

- Performance data contained herein was generally obtained in a controlled, isolated environments. Customer examples are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual performance, cost, savings or other results in other operating environments may vary.

- References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business.

- Workshops, sessions and associated materials may have been prepared by independent session speakers, and do not necessarily reflect the views of IBM. All materials and discussions are provided for informational purposes only, and are neither intended to, nor shall constitute legal or other guidance or advice to any individual participant or their specific situation.

- It is the customer's responsibility to insure its own compliance with legal requirements and to obtain advice of competent legal counsel as to the identification and interpretation of any relevant laws and regulatory requirements that may affect the customer's business and any actions the customer may need to take to comply with such laws. IBM does not provide legal advice or represent or warrant that its services or products will ensure that the customer follows any law.

AGENDA

Note: Some “bonus” material on the row organized table layout (and indexes too) is included after the last main slide!

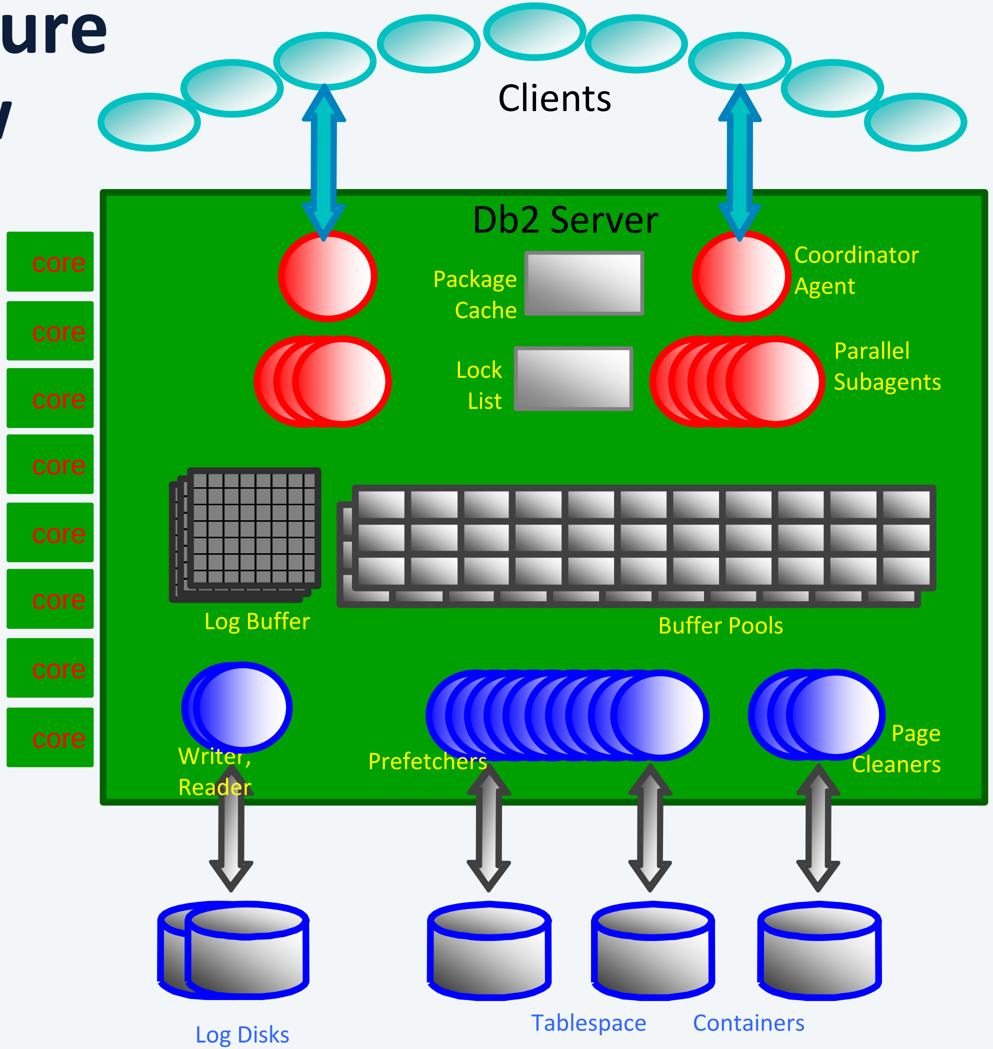
Overview :

- Architecture Overview
  - Basic Operation Walkthroughs (Row and Columnar)

BLU:

- What is Db2 with BLU Acceleration?
- Working with Databases and Column-Organized Tables
- BLU Query Processing
- Columnar Compression and Storage
- What’s new in v11.5 and beyond?
- Best practices, Tips, and Tricks

Architecture Overview



Parallelism

- SQL and Utilities
- Intra- & Intra-Partition Parallelism
- Cost-based Optimizer with Query Rewrite

Multi-core Exploitation

- All cores exploited through Operating System threads

Very Large Memory Exploitation

- 64 bit Support
- I/O Buffering
- Multiple Buffer Pools

I/O Subsystem

- Asynchronous, Parallel I/O
- Automatic, Intelligent Data Striping with Parallel I/O
- Big block I/O
- Scatter/Gather I/O

Database Partitioning Feature

Shared Nothing Architecture Allows Virtually Unlimited Scalability

- Each partition owns it's resources (buffer pool, locks, disks,...)
- Avoids common limits on scalability:
  - No need for distributed lock manager or buffer coherence protocols
  - No need to attach disks to multiple machines
- Partitions Communicate Only Necessary Tuples
  - Using shared memory (same machine)
  - Using high speed comm (diff. machines)

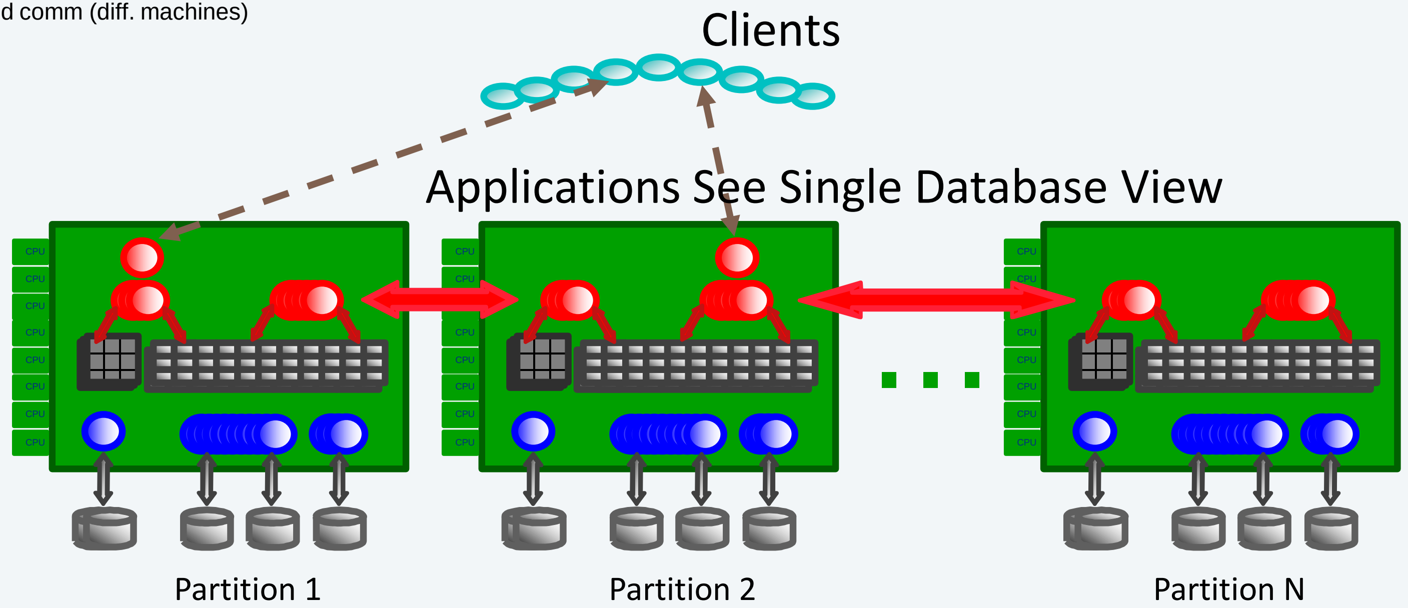
Partitions are Logical

- Any number of partitions can be created on a single physical machine (works extremely well with NUMA architectures)

Virtually Everything Runs in Parallel Across Nodes

- SQL: queries, inserts, updates, deletes
- Utilities: Backup, Restore, Load, Index Create, Reorg

Near Linear Scaling For Warehousing



Architecture Overview : pureScale

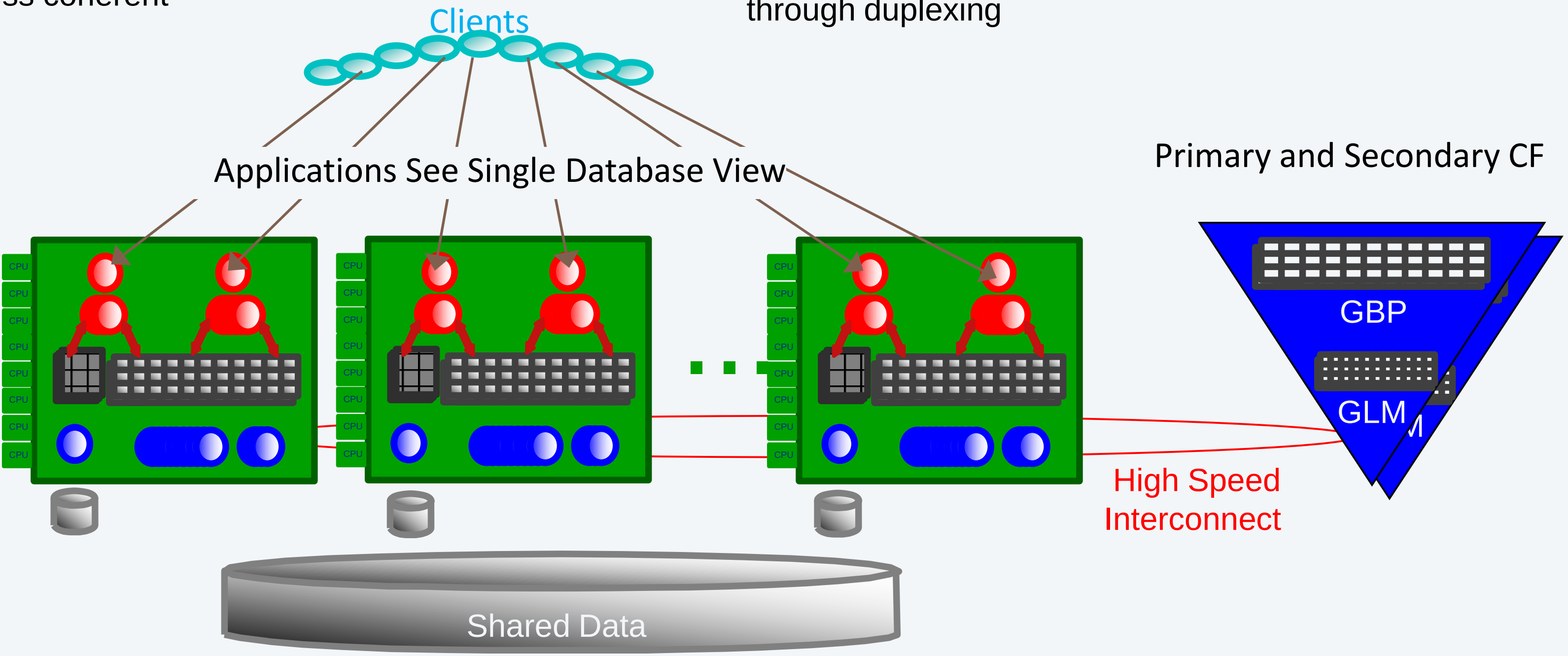
Shared Data Architecture

- Members have equal access to database storage
- Clients connect to any member and get completely coherent data access
- Members co-operate with each other and the CF to keep data concurrent data access coherent
- Per-member logs

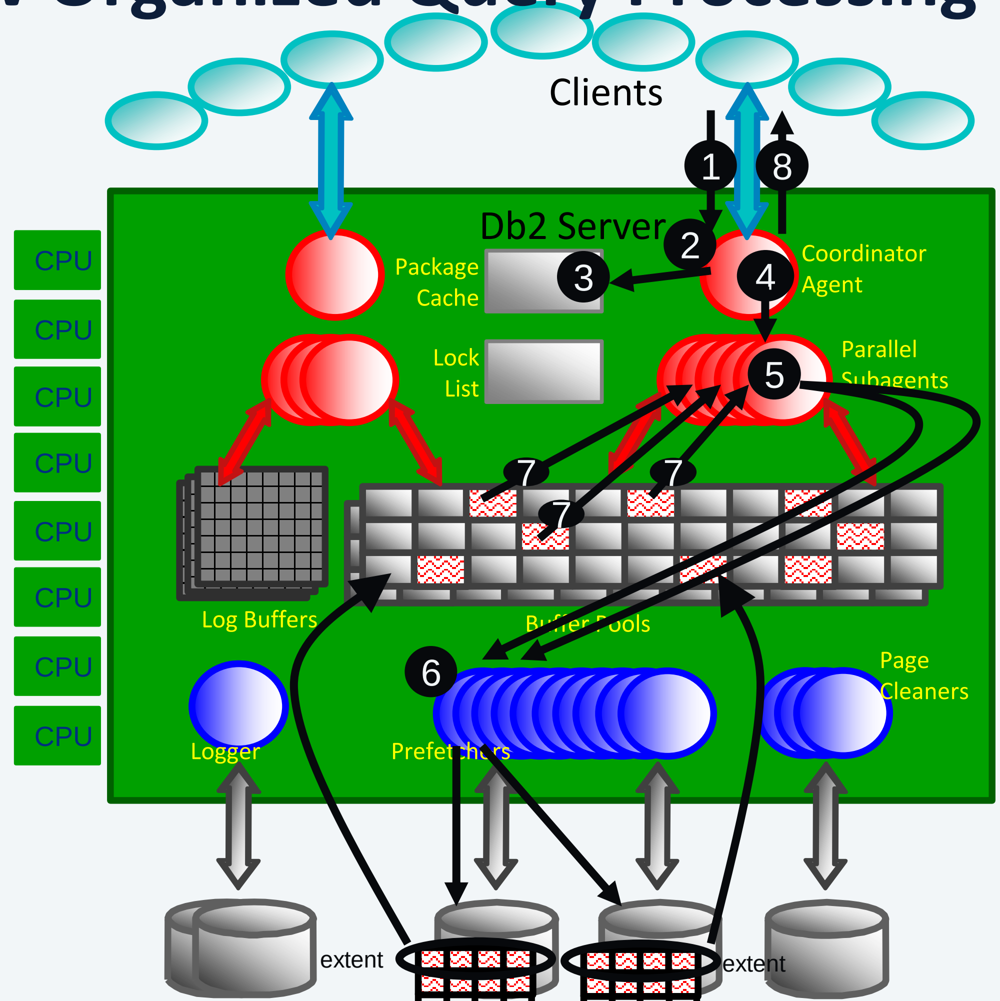
Cluster Caching Facility (CF)

- Provides a global lock manager (GLM)
- Provides another level of buffer pool (GBP) above disk
- Redundant CFs kept in-sync with each other through duplexing

Near Continuous Availability for OLTP



Row Organized Query Processing

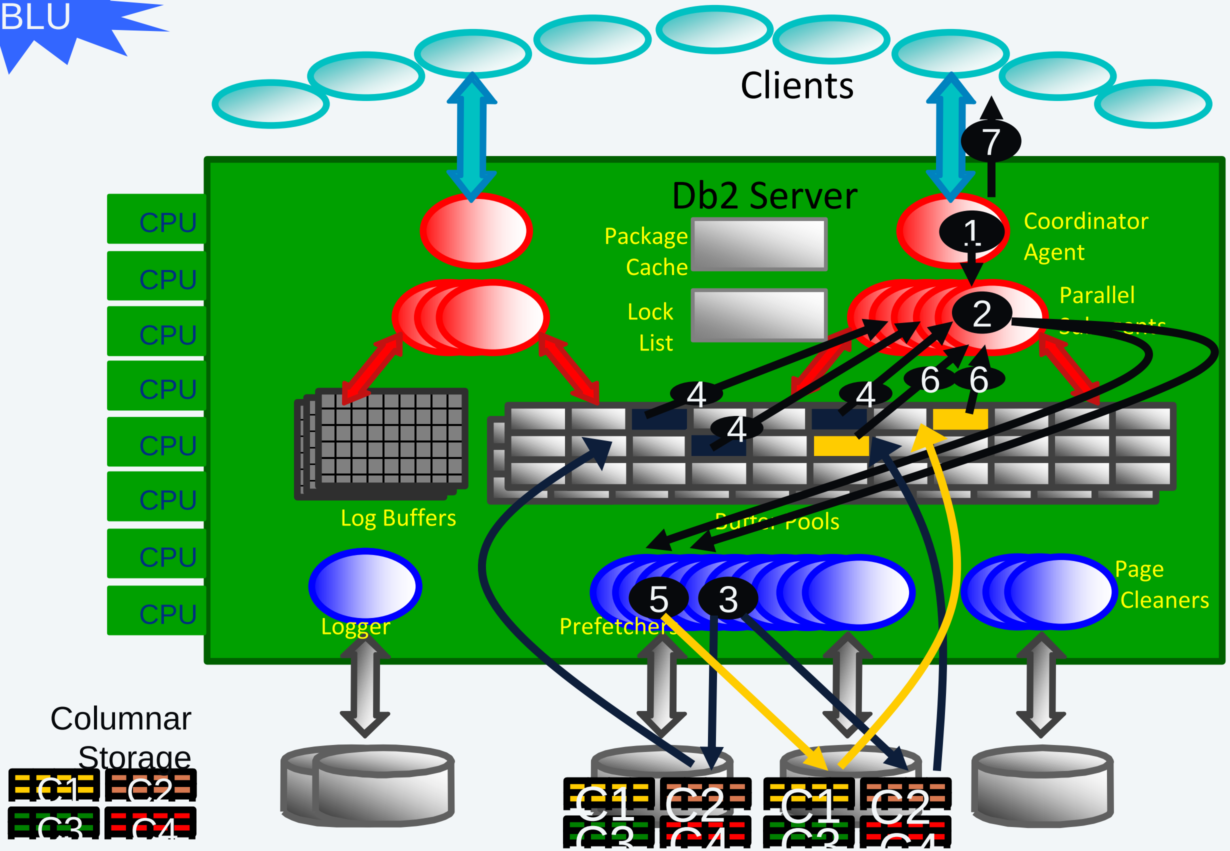


SELECT C1 FROM T1 WHERE C2='5'

1. SQL statement sent over network to coordinator agent
2. SQL statement compiled and optimized
3. Resulting access plan stored in shared access plan cache
4. Access plan execution begins; subagents perform parallel table scan
5. Periodic async prefetch requests sent to prefetchers (aka 'io servers')
6. Prefetchers asynchronously drive parallel I/O against tablespace containers to bring in extents from disk into separate pages in bufferpool
7. Entire rows read out of buffer pool and decompressed. C2 values compared to '5'. Matching C1 values added to result set.
8. Result set sent back to client.

Column Organized Query Processing

BLU



Each extent contains values for 1 column

(*) Synopsis filtering not shown here; More on this later.

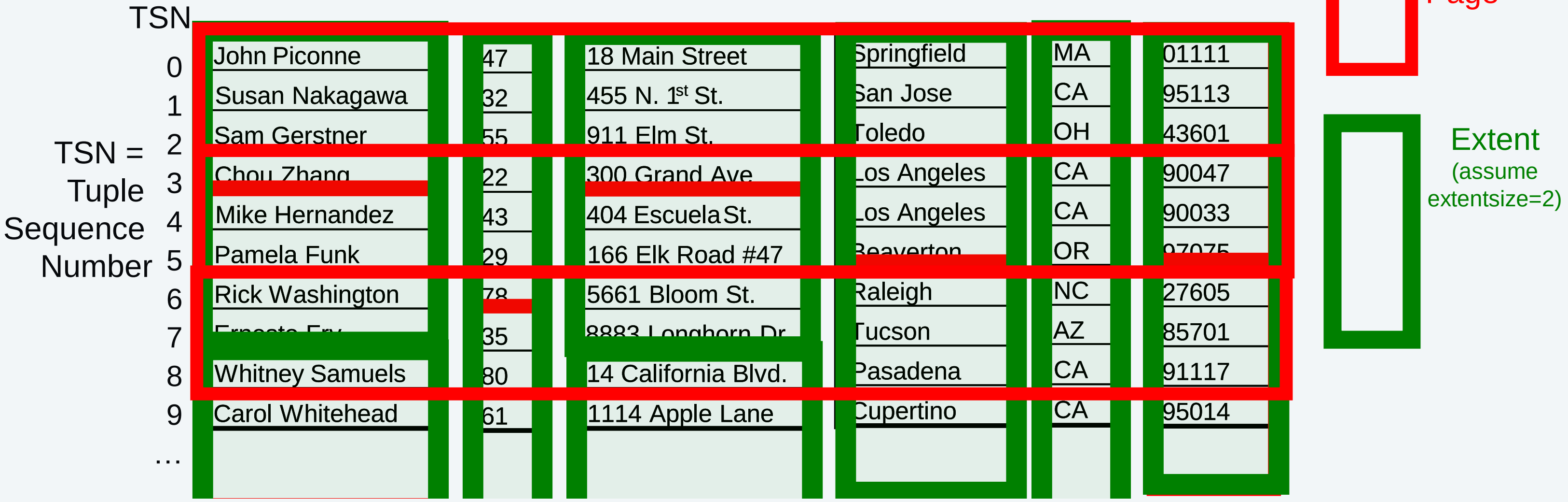
SELECT C1 FROM T1 WHERE C2=‘5’

… Initial steps skipped …

1. Access plan execution begins; subagents kicked off to perform parallel scan of column C2
2. Periodic prefetch requests sent to prefetchers (aka ‘io servers’)
3. Prefetchers asynchronously drive parallel I/O against tablespace containers to bring requested pages containing only C2 values from disk into separate pages in bufferpool (*)
4. Batch of C2 values is read out of buffer pool and compared to ‘5’ (data remains compressed - “active” compression), forming a batch of qualifying tuple sequence numbers (TSNs)
5. The C1 values corresponding to the batch of qualifying TSNs are prefetched
6. Qualifying C1 values added to result set,…
7. and sent to client

Deeper Look at Internals : Column Storage

- With traditional tables, each page contains entire rows
- With BLU, each page and extent contains values for a single column



TSN						
0	John Piconne	47	18 Main Street	Springfield	MA	01111
1	Susan Nakagawa	32	455 N. 1st St.	San Jose	CA	95113
2	Sam Gerstner	55	911 Elm St.	Toledo	OH	43601
3	Chou Zhang	22	300 Grand Ave	Los Angeles	CA	90047
4	Mike Hernandez	43	404 Escuela St.	Los Angeles	CA	90033
5	Pamela Funk	29	166 Elk Road #47	Beaverton	OR	97075
6	Rick Washington	78	5661 Bloom St.	Raleigh	NC	27605
7	Ernesto Fry	35	8883 Longhorn Dr.	Tucson	AZ	85701
8	Whitney Samuels	80	14 California Blvd.	Pasadena	CA	91117
9	Carol Whitehead	61	1114 Apple Lane	Cupertino	CA	95014
…						

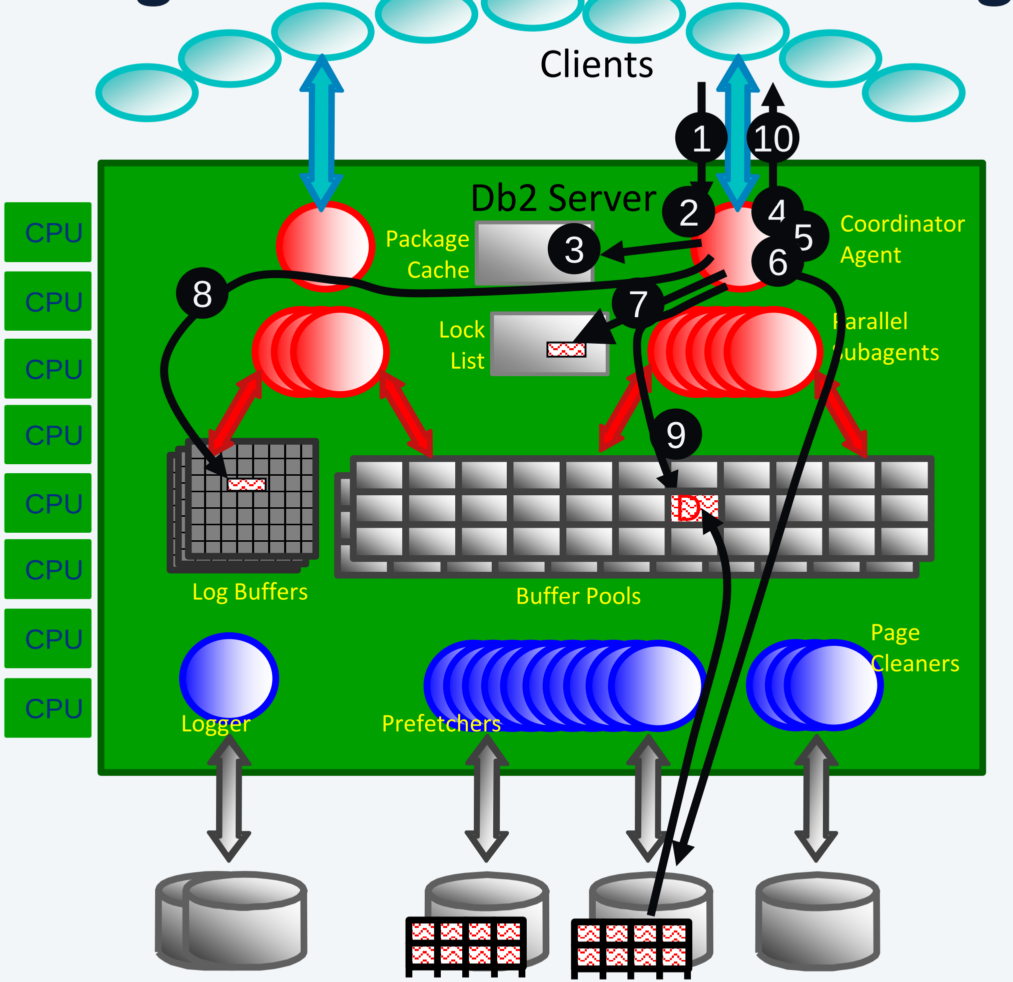
TSNs (a logical Row ID) are used to stitch together column values that belong in the same row during query processing

- eg. SELECT zipcode FROM t WHERE name="Mike Hernandez"
  - an internal 'page map index' allows Db2 to quickly find the page containing the zipcode for TSN 4

Typically, column-organized tables use significantly less space than row-organized tables

- Unusual case: column-organized tables with many columns and very few rows can be larger than row-organized tables as each column requires at least 1 extent

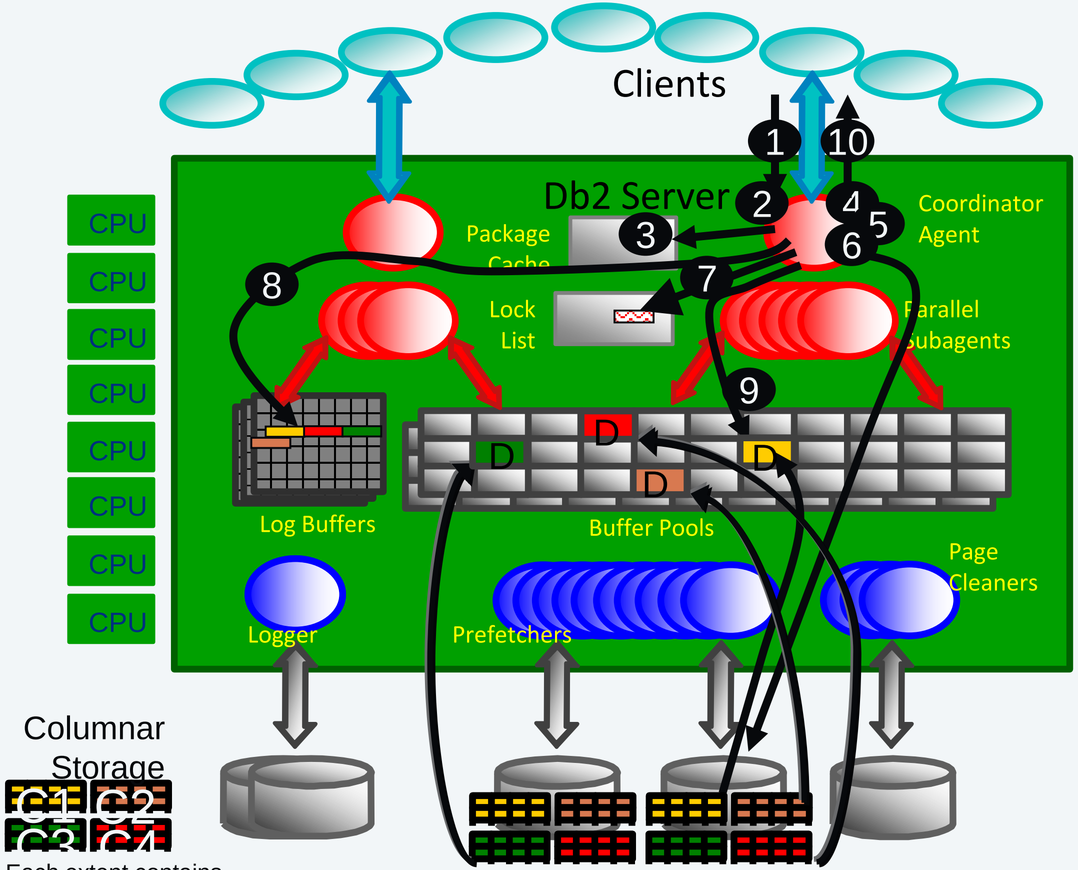
Row Organized INSERT Processing



INSERT INTO T1 (…)

1. SQL statement sent over network to coordinator agent
2. SQL statement compiled and optimized
3. Resulting access plan stored in shared access plan cache
4. Access plan execution begins
5. Agent searches for a page in the table large enough for row
6. Page found, and read into buffer pool
7. Agent acquires X lock on row
8. Agent writes log record to log buffer in memory (describes how to redo and undo the upcoming insert)
9. Agent inserts record to page in buffer pool (“dirties” page)
10. Success sent to client

Column Organized INSERT Processing



(*) Synopsis maintenance not shown here; More on this later.

INSERT INTO T1 (…)

1. SQL statement sent over network to coordinator agent
2. SQL statement compiled and optimized
3. Resulting access plan stored in shared access plan cache
4. Access plan execution begins
5. For each column, agent finds a page large enough for the column value (BLU uses an append approach for this)
6. Pages found, and read into buffer pool
7. Agent acquires X lock on logical row (aka TSN)
8. For each column, agent writes log record to log buffer in memory (describes how to redo and undo per column)
9. Agent inserts column values to pages in buffer pool ("dirties" pages)
10. Success sent to client

Nothing Written to Disk during the Insert ??

Db2 tries to do expensive operations like I/Os in batches and in the background as much as possible. Why? To optimize:

- Overall system throughput
- Individual statement response time

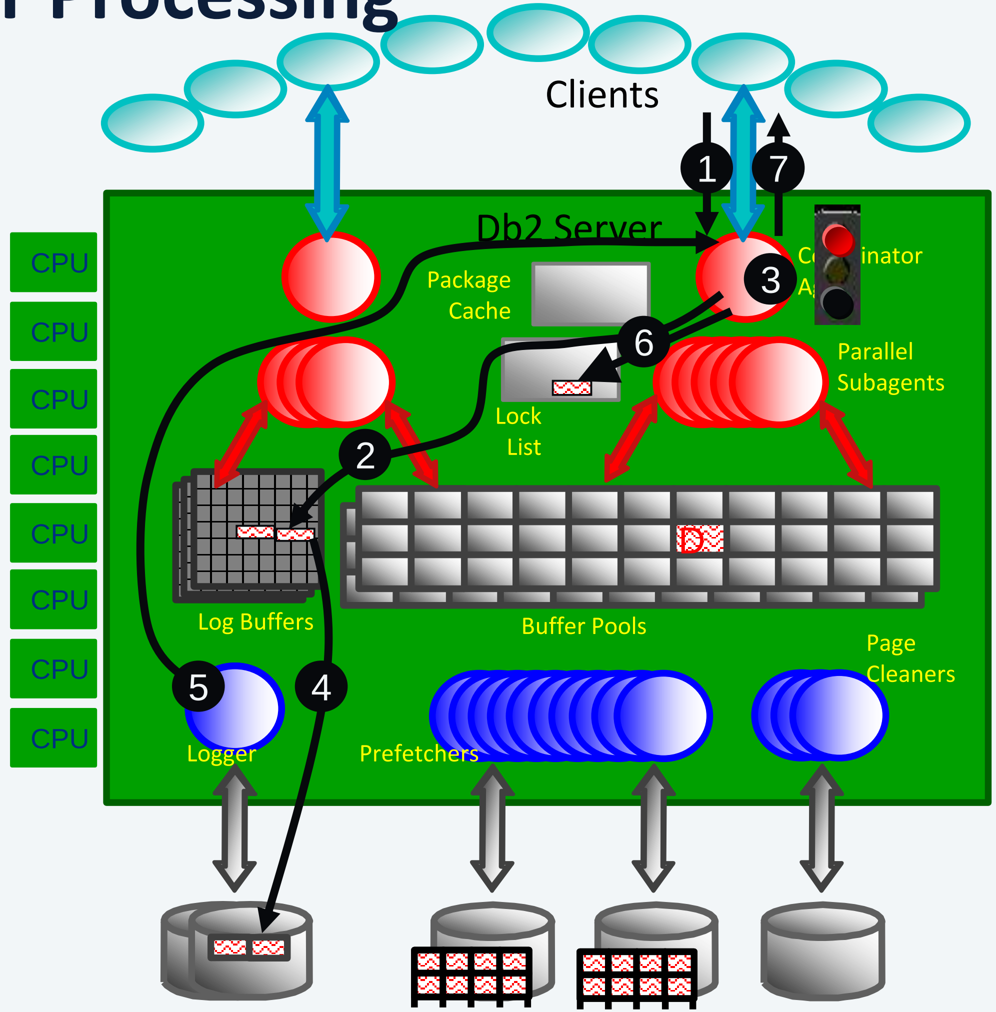
In this case, the inserting transaction is not yet committed

- So, there’s no fundamental need to write the inserted row to disk
  - Other transactions will retrieve the latest value from the buffer pool
  - And, if the system crashes, the database must show the insert transaction as having not occurred

When *should* DB2 write the following to disk ?

- The log record in the log buffer ?
- The dirty data page with the new row in the buffer pool ?

COMMIT Processing



1. COMMIT SQL statement sent over network to coordinator agent

2. Agent writes commit log record to log buffer

3. Agent waits for logger to write log buffer (up to and including the commit log record) to disk (if not already done)

4. Logger "gets around" to writing needed log buffers to log disk (at this point, the transaction is durable)

5. Logger posts all agents that are waiting for 'hardening' of the log records just written to the log disk

6. Lock released

7. Success sent to client

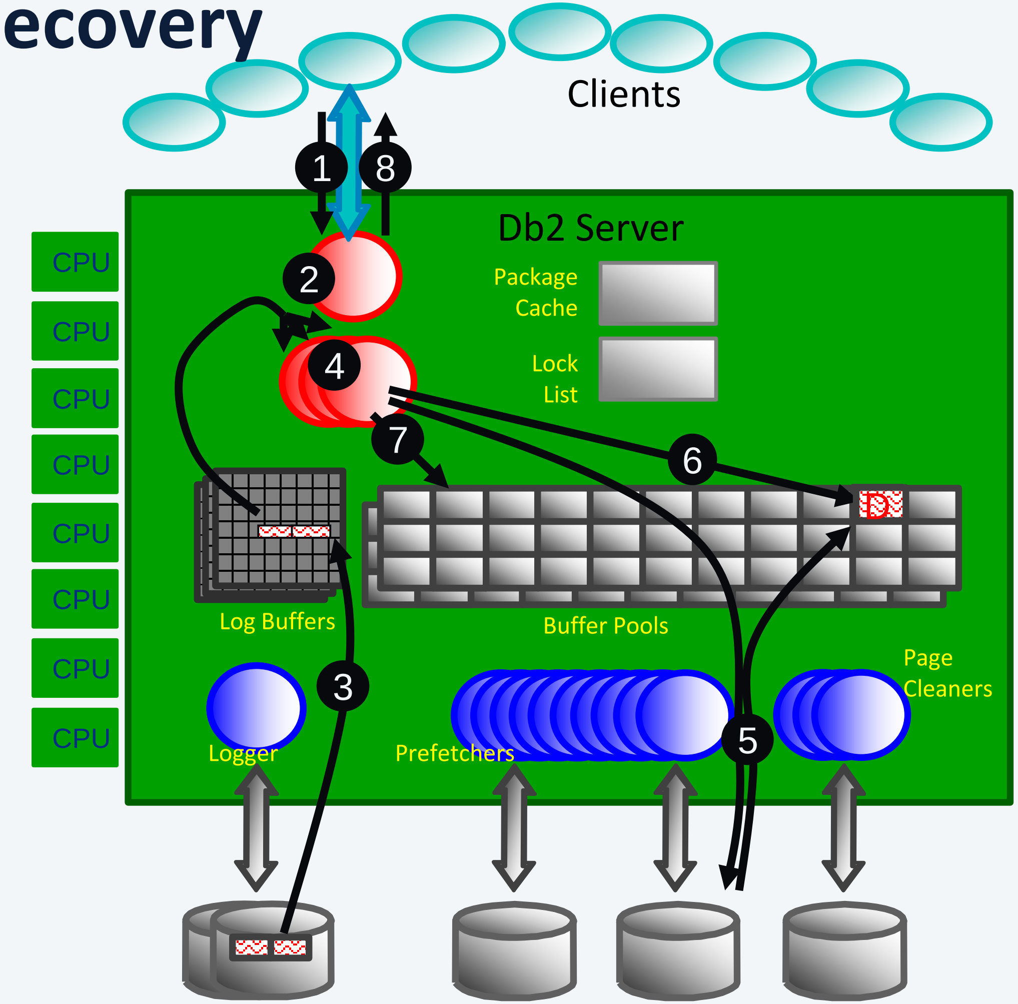
What About the Dirty Data Page ?

When does it get written to disk ?

What happens if power is lost *right now ?*

- Will the committed insert be lost ?
- How is it recovered ?

Crash Recovery

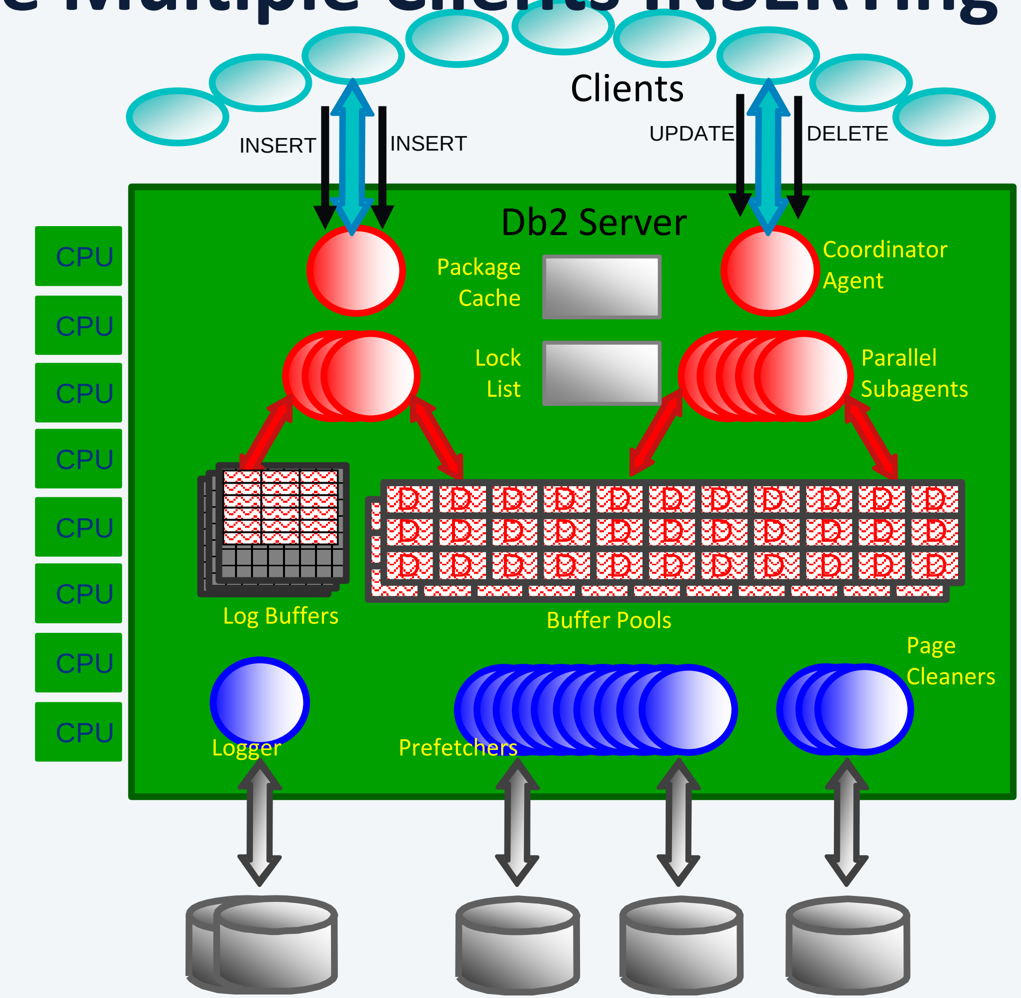


1. Client tries to CONNECT, RESTART or ACTIVATE the database
2. Agent realizes database is in inconsistent state so initiates crash recovery
3. Log reader reads "active" log records into the log buffer
4. Subagents "redo" log records in parallel
5. For each log record, read target page into buffer pool, and,…
6. … redo the action specified in the log record (if it's not already reflected in the page)
7. After redo phase, the "undo" phase will undo any actions done for transactions that did not commit before the system crash
8. When redo and undo complete, the database is open for other clients, and success is returned

OK, But What About That Dirty Data Page ?

When *does* it get written to disk ?

Imagine Multiple Clients INSERTing (or U/D)

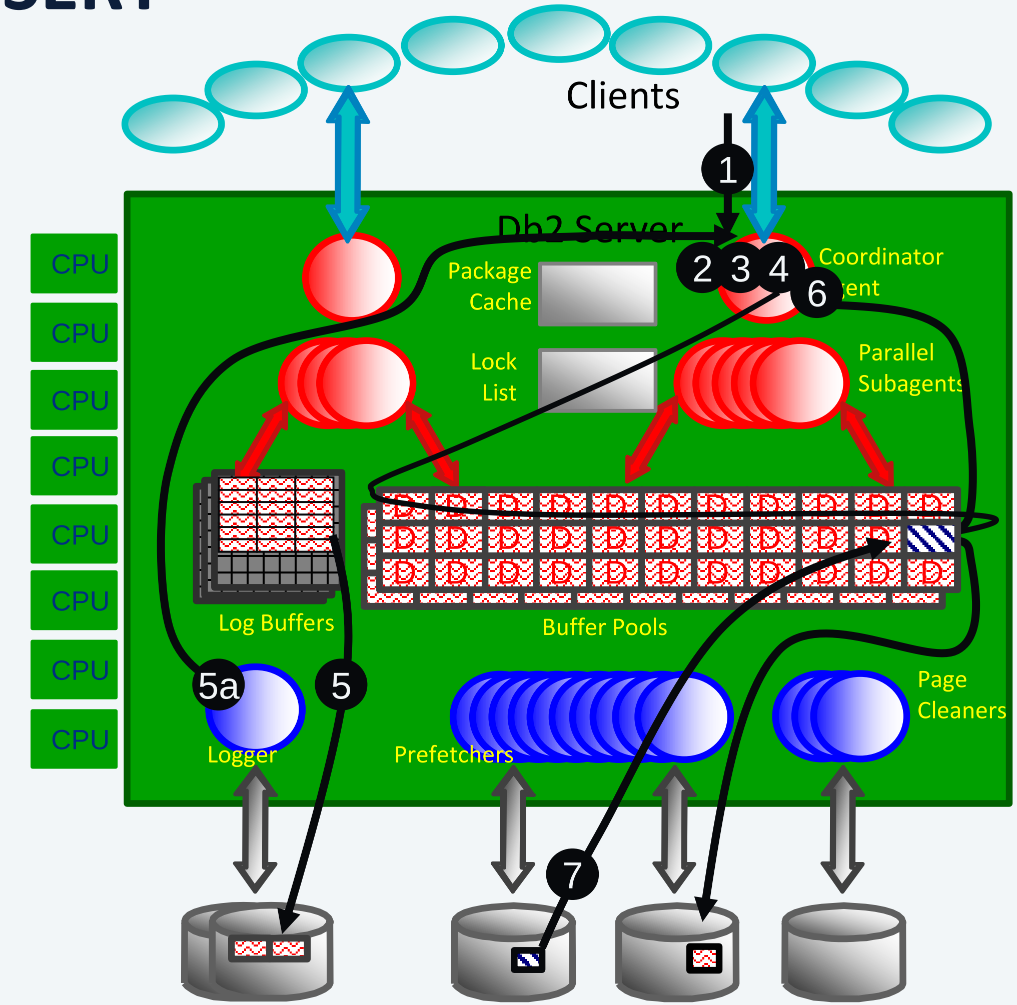


The buffer pool is full of dirty pages.

What happens when an agent tries to insert to (yet) another page ?

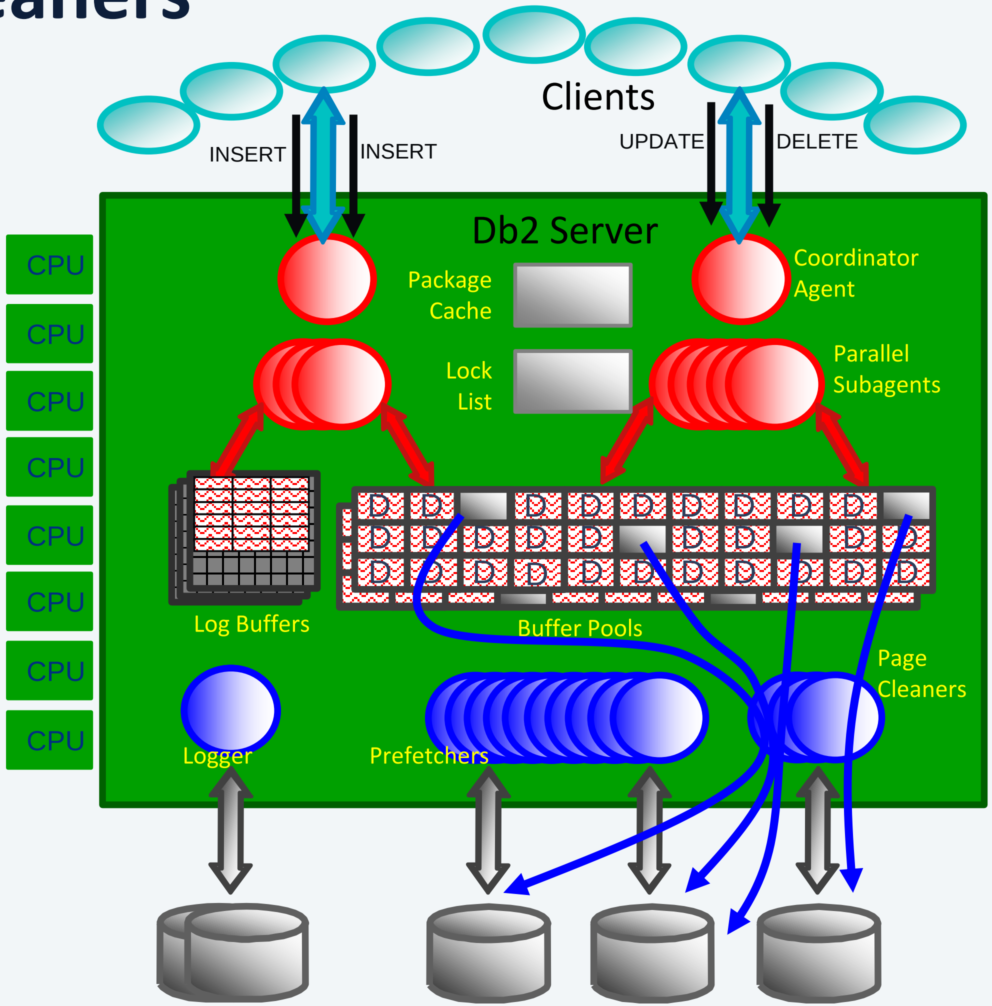
Next INSERT

“WAL” & “Dirty Steals”



1. INSERT SQL statement sent over network to coord agent
2. (Skip SQL compilation, optimiz’n, access plan mgt, etc)
3. Agent finds a page with enough space, and tries to read it into the buffer pool
4. Buffer pool manager chooses a ‘victim’ page. It tries to choose a clean LRU page using ‘clock’ algorithm. If can’t find a clean page, will choose dirty victim - a ‘dirty steal’.
5. Dirty victim must be written to disk. However, before that can be done, associated log record must be written to log disk (Why?). Note, this policy is called “WAL” (or “write ahead logging”). 5a: logger posts interested agents.
6. Now agent writes dirty victim page to disk.
7. Now (finally) target page can be read into buffer pool and updated.

Page Cleaners



Write dirty pages to disk in the background.

Why ??
… Performance

- *Insert/Update/Deletes don't wait*
- *More efficient batch I/O*
- *Avoid Dirty Steals*

AGENDA

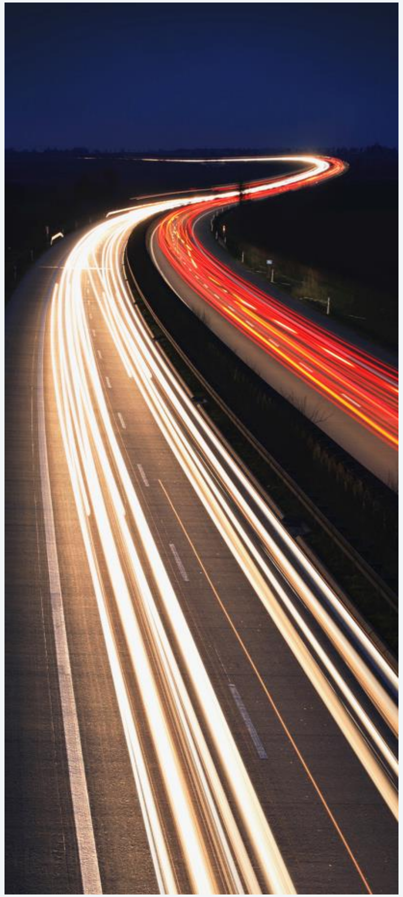
Overview :

- Architecture Overview
  - Basic Operation Walkthroughs (Row and Columnar)

BLU (columnar):

- What is Db2 with BLU Acceleration?
- Working with Databases and Column-Organized Tables
- BLU Query Processing
- Columnar Compression and Storage
- What’s new in v11.5 and beyond?
- Best practices, Tips, and Tricks

What is Db2 with BLU Acceleration?



Next generation database for analytics

- Performance improvements
- Storage savings
- Simplicity

Seamlessly integrated

- Built directly into Db2
- Consistent SQL, language interfaces, administration

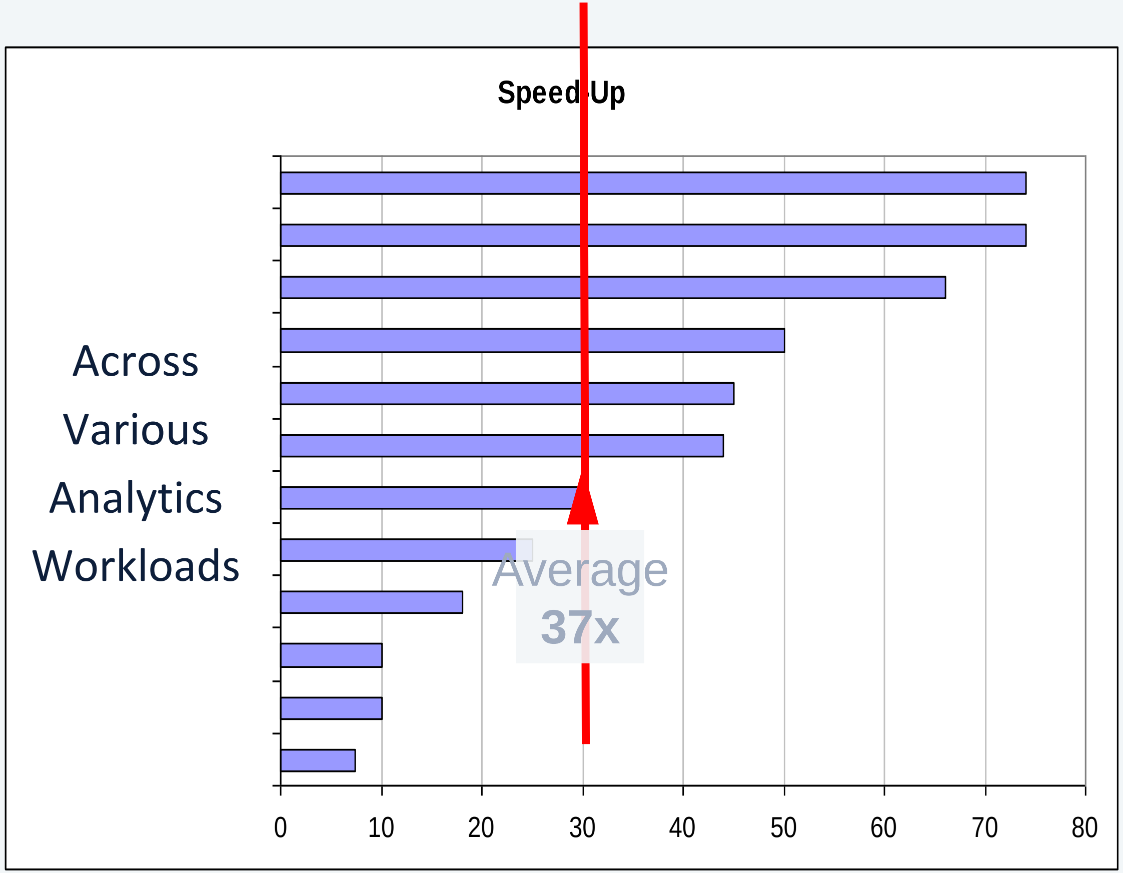
Hardware Optimized

- Memory, CPU, and I/O optimized

Available in Db2 on-prem, Db2W, Db2WoC, and IIAS

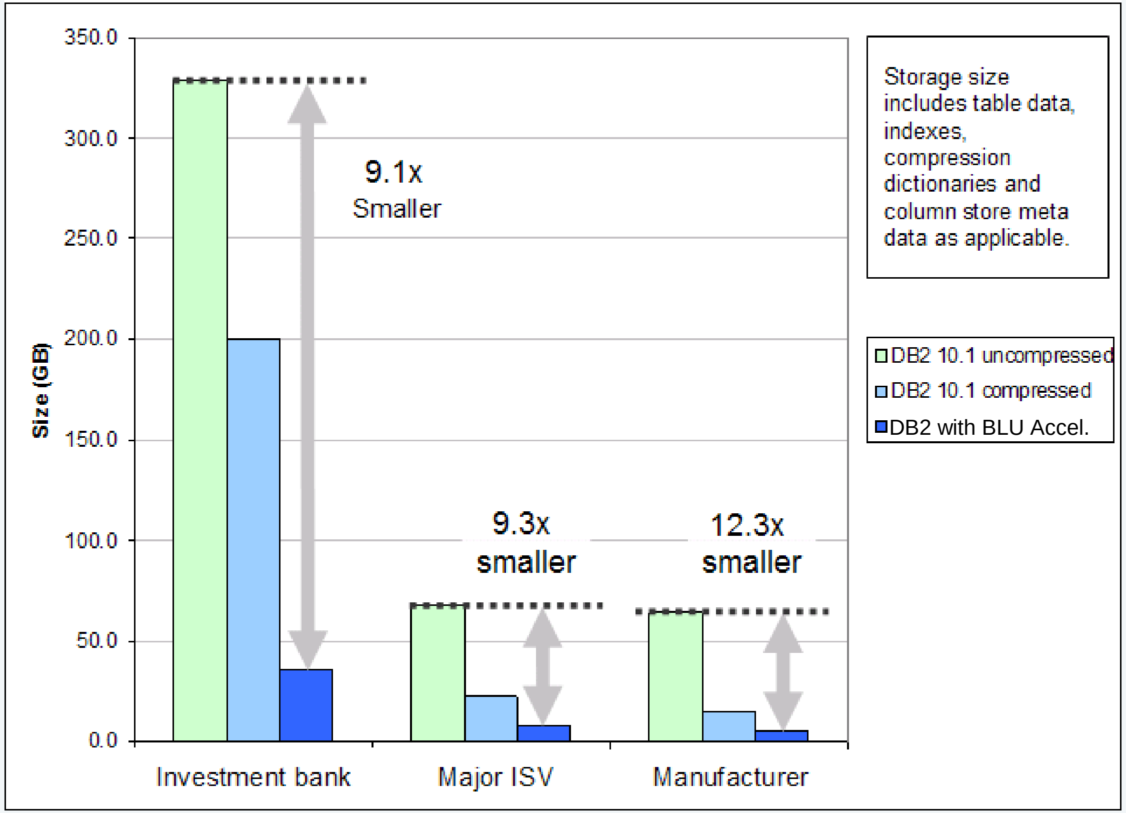
Super Fast: Query Speed-up Examples

- Significant speed-up for many warehouse workloads
- Identical hardware vs. traditional row-based analytic database technology
- Some queries 1000+x faster



Super Small: 10x Storage Reduction Common

- 10x or more compression commonly reported
- A further 2x-3x storage reduction vs. Db2’s previous industry-leading adaptive compression



Super Easy

Database Design and Tuning

VS

1. Decide on partition strategies
2. Select Compression Strategy
3. Create Table
4. Load data
5. Create Auxiliary Performance Structures
   - Materialized views
   - Create indexes
     - B+ indexes
     - Bitmap indexes
6. Tune memory
7. Tune I/O
8. Add Optimizer hints
9. Statistics collection

Repeat

DB2 with BLU Acceleration

1. Create Table
2. Load data

Will your Workload Benefit from BLU Acceleration?

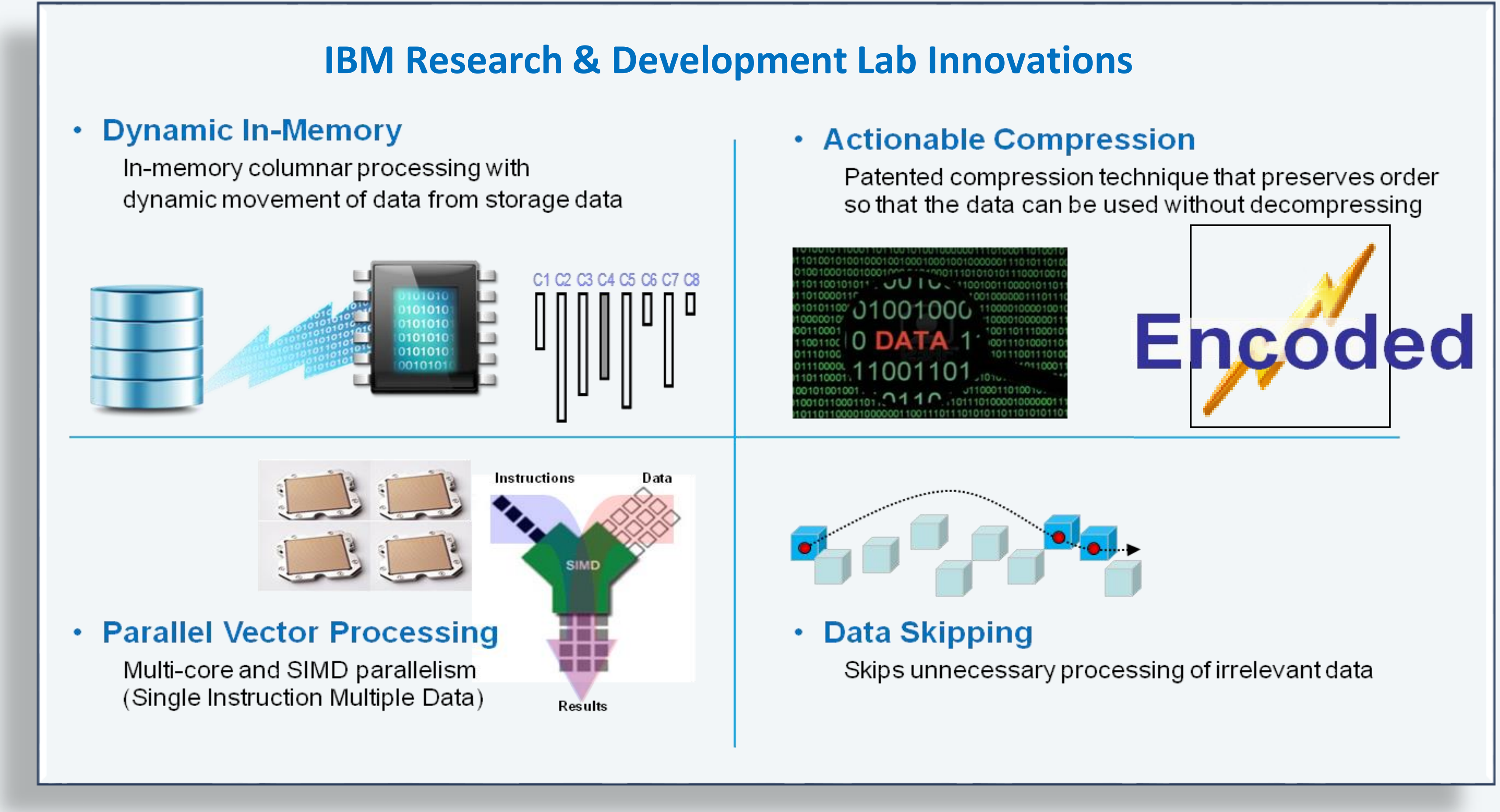
Probably:

- Analytical workloads, data marts, etc.
- Grouping, aggregation, range scans, joins
- Queries touch only a subset of the columns in a table
- Star Schema
- SAP Business Warehouse
- Netezza migration

Probably not:

- OLTP
- Insert, Update, Delete of few rows per transaction*
- Queries touch many or all columns in a table
- Use of XML, pureScale, etc. which are not supported in BLU yet

Db2 BLU Core Concepts



IBM Research & Development Lab Innovations

- **Dynamic In-Memory**
  In-memory columnar processing with dynamic movement of data from storage data

- **Actionable Compression**
  Patented compression technique that preserves order so that the data can be used without decompressing

- **Parallel Vector Processing**
  Multi-core and SIMD parallelism (Single Instruction Multiple Data)

- **Data Skipping**
  Skips unnecessary processing of irrelevant data

Creating a Database for BLU Acceleration

- Step 1: Set DB2_WORKLOAD registry variable for optimal configuration defaults
  db2set DB2_WORKLOAD=ANALYTICS
  - This setting is used by AUTOCONFIGURE to influence default configuration and optimize for BLU Acceleration analytic workloads
  - Don’t disable AUTOCONFIGURE
- Step 2: Create your database
  - Refer to Notes for an example plus important extra settings.

Optimizing an Existing Database for BLU Acceleration

- Step 1: Set DB2_WORKLOAD registry variable for optimal configuration defaults
  db2set DB2_WORKLOAD=ANALYTICS

- Step 2: Run AUTOCONFIGURE to get most of the recommended settings

DB2_WORKLOAD=ANALYTICS Sets Everything You Need

dft_table_org = COLUMN
default page size for new DB is 32K,   dft_extent_sz = 4
dft_degree = ANY,   intra-query parallelism is enabled

catalogcache_sz – higher value than default

sortheap and sheapthres_shr – higher value than default
util_heap_sz – higher value than default

WLM controls concurrency on SYSDEFAULTMANAGEDSUBCLASS
Automatic table maintenance and auto_reorg = ON
* And more!

Creating a Column-Organized Table

```
CREATE TABLE sales_col (
   c1 INTEGER NOT NULL,
   c2 INTEGER,
   . . .
   PRIMARY KEY (c1) ) ORGANIZE BY COLUMN;
```

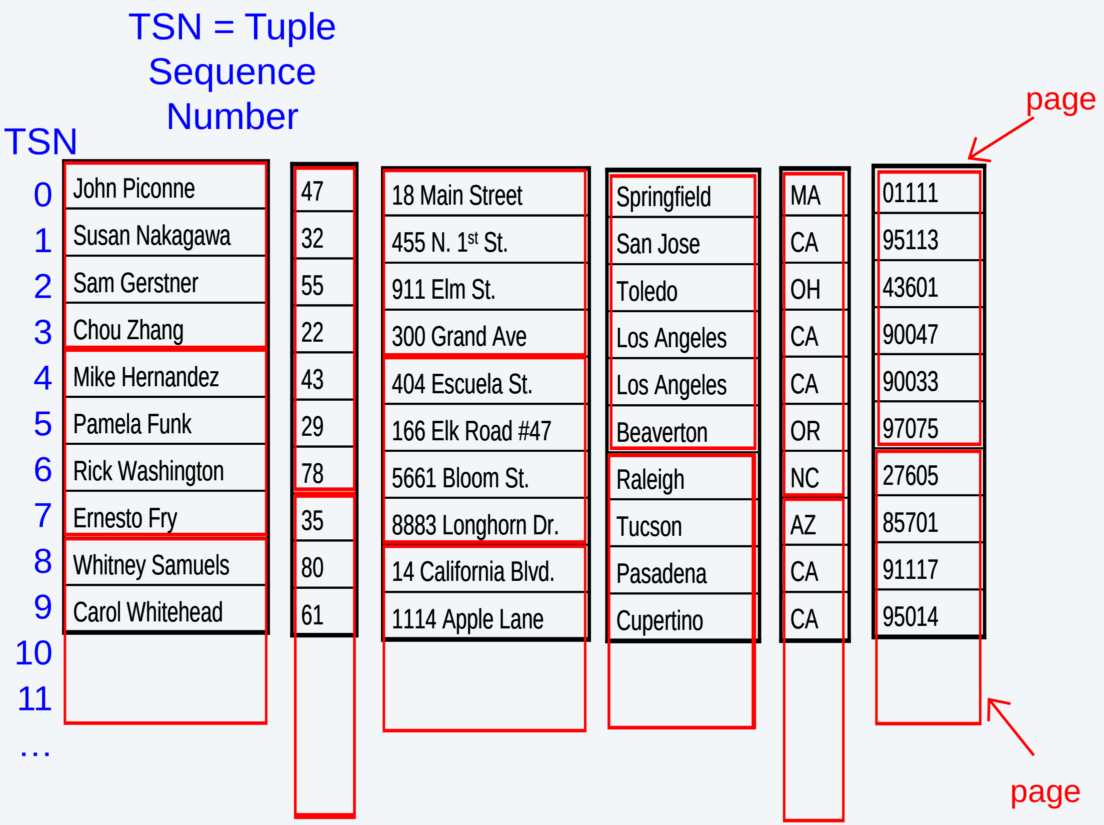
Columnar tables are always compressed by default.

- If dft_table_org = COLUMN
  - ORGANIZE BY COLUMN is the default and can be omitted
  - Use ORGANIZE BY ROW to create row-organized tables

Columnar Storage in Db2 (Conceptual)

- Separate set of extents and pages for each column

- Typically, column-organized tables use less space than row-organized tables



TSN = Tuple Sequence Number

page

page

TSN						
0	John Piconne	47	18 Main Street	Springfield	MA	01111
1	Susan Nakagawa	32	455 N. 1st St.	San Jose	CA	95113
2	Sam Gerstner	55	911 Elm St.	Toledo	OH	43601
3	Chou Zhang	22	300 Grand Ave	Los Angeles	CA	90047
4	Mike Hernandez	43	404 Escuela St.	Los Angeles	CA	90033
5	Pamela Funk	29	166 Elk Road #47	Beaverton	OR	97075
6	Rick Washington	78	5661 Bloom St.	Raleigh	NC	27605
7	Ernesto Fry	35	8883 Longhorn Dr.	Tucson	AZ	85701
8	Whitney Samuels	80	14 California Blvd.	Pasadena	CA	91117
9	Carol Whitehead	61	1114 Apple Lane	Cupertino	CA	95014
10						
11						
…						

What You See in the Db2 Catalog: TABLEORG

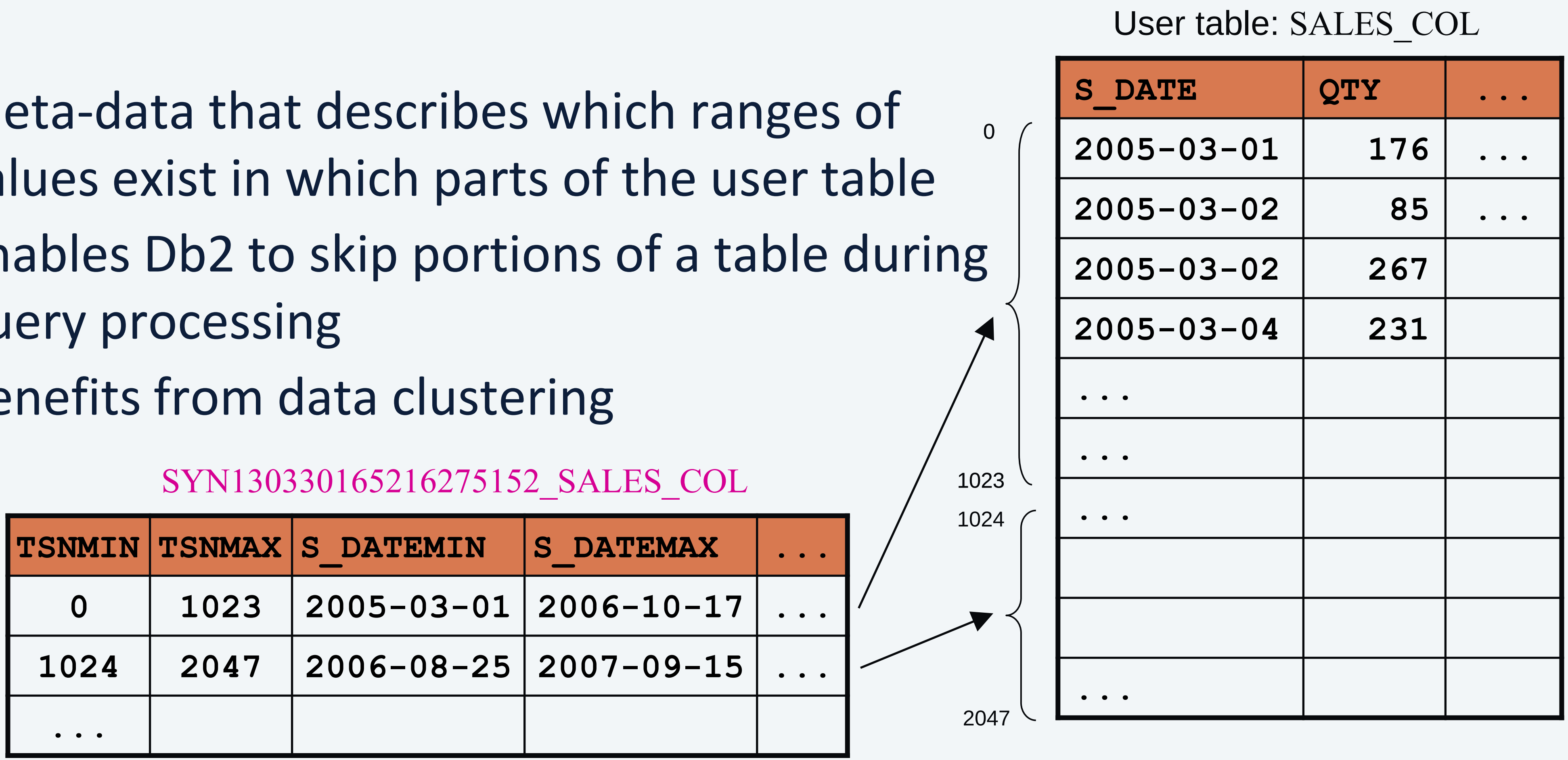
- Which tables are column-organized?
  - New column in syscat.tables: TABLEORG

```
SELECT tabname, tableorg, compression
FROM   syscat.tables
WHERE  tabname like 'SALES%';

TABNAME                          TABLEORG COMPRESSION
-------------------------------- -------- -----------
SALES_COL                        C
SALES_ROW                        R        N

  2 record(s) selected.
```

For column-organized tables, COMPRESSION is always blank because you cannot enable/disable compression.

Synopsis Table - data skipping

- Meta-data that describes which ranges of values exist in which parts of the user table
- Enables Db2 to skip portions of a table during query processing
- Benefits from data clustering



SYN130330165216275152_SALES_COL

TSNMIN	TSNMAX	S_DATEMIN	S_DATEMAX	...
0	1023	2005-03-01	2006-10-17	...
1024	2047	2006-08-25	2007-09-15	...
...				

User table: SALES_COL

S_DATE	QTY	...
2005-03-01	176	...
2005-03-02	85	...
2005-03-02	267	
2005-03-04	231	
...		
...		
...		
...		

What You See in the Db2 Catalog: Synopsis Tables

- For each column-organized table there is a corresponding *synopsis table*, automatically created and maintained.

```
SELECT tabschema, tabname, tableorg
FROM syscat.tables
WHERE tableorg = 'C';

TABSCHEMA       TABNAME                            TABLEORG
--------------- ---------------------------------- --------
CDREXELI        SALES_COL                          C
SYSIBM          SYN130330165216275152_SALES_COL    C

  2 record(s) selected.
```

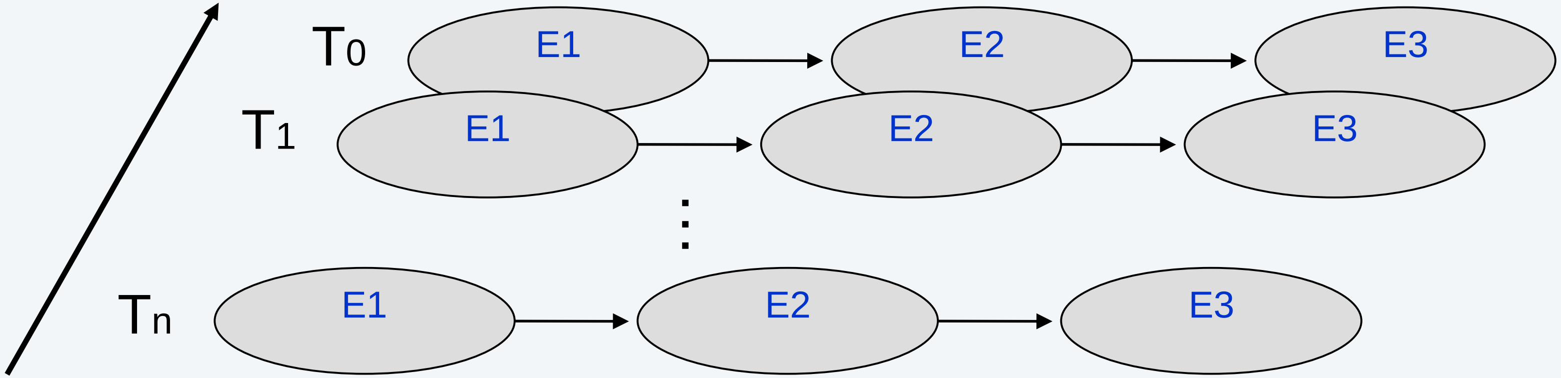
What You See in the Db2 Catalog: Page Map Index

- Automatically created and maintained.
- Used internally to locate column data in storage object.
- Maps columns and TSNs to data pages.

```
SELECT indschema, indname, colnames, indextype
FROM syscat.indexes
WHERE tabname = 'SALES_COL';

INDSCHEMA  INDNAME             COLNAMES           INDEXTYPE
---------- ------------------- ------------------ ---------
SYSIBM     SQL130330165215840  +ID                REG
SYSIBM     SQL130330165216790  +COLGID+STARTTSN   CPMA

  2 record(s) selected.
```

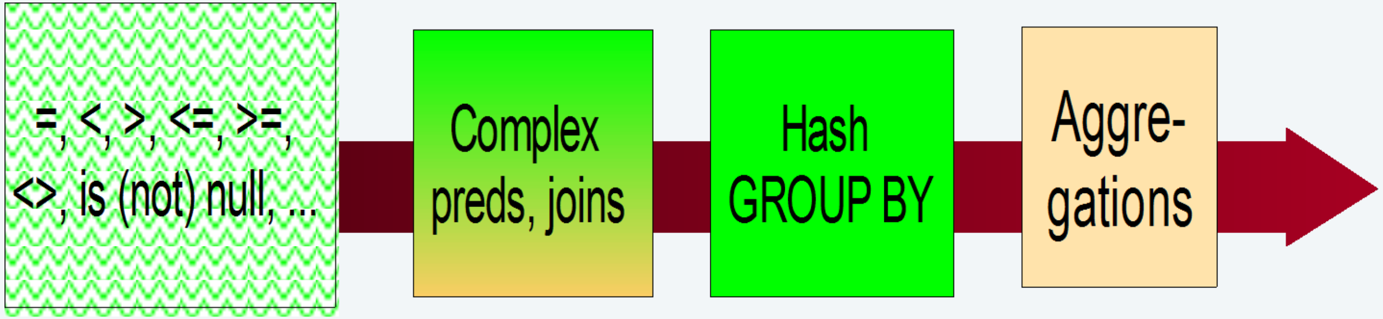
BLU Query Runtime Execution Flow

- Evaluator chains contain specific evaluators for different operations
- Multiple DB sub-agents execute cloned evaluator chains in parallel
  - Threading degree determined by the optimizer
- Straw model to distribute TSN ranges



BLU Query Processing Order

- Synopsis scan to skip tuples
- Predicates on compressed data
- Join and group-by
- Apply after decoding
  - Complex expressions, arithmetic, aggregations



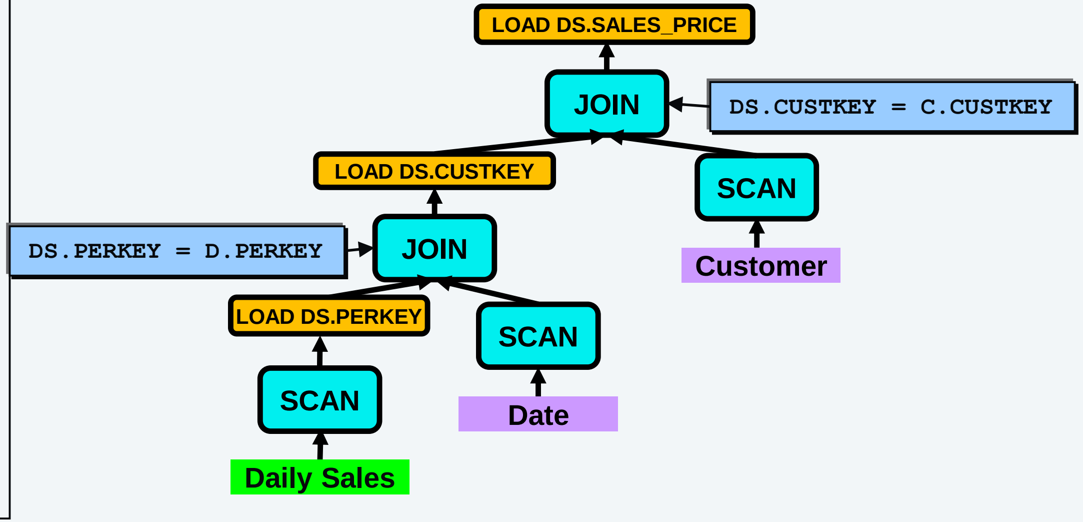
BLU Query Optimization

- Late materialization
  - Columns are retrieved as late as possible depending on predicate filtering
  - Occurs for TBSCANs and probe side of HSJOINs
    - e.g. SELECT C1, C2, C3 FROM T1 WHERE C1=5 AND C2=10
      - SCAN C1, apply C1=5, return row-ids
      - SCAN C2, using row IDs from 1), apply C2=10, return row IDs
      - SCAN C3, using row IDs from 2), return values
  - Determined dynamically by BLU runtime
  - Accounted for in the optimizer’s cost model

Late materialization

- For HSJOIN probe side:
  - Retrieve columns needed for join just before the join
  - Retrieve columns not required for predicate application, after all joins have been performed

```
select
          c.first_name,
          c.last_name,
          ds.sales_price
from
           customer c,
          date d,
           daily_sales ds
where
  ds.perkey = d.perkey and
  ds.custkey = c.custkey and
  d.year = 2015
```



Actionable Compression 2/2

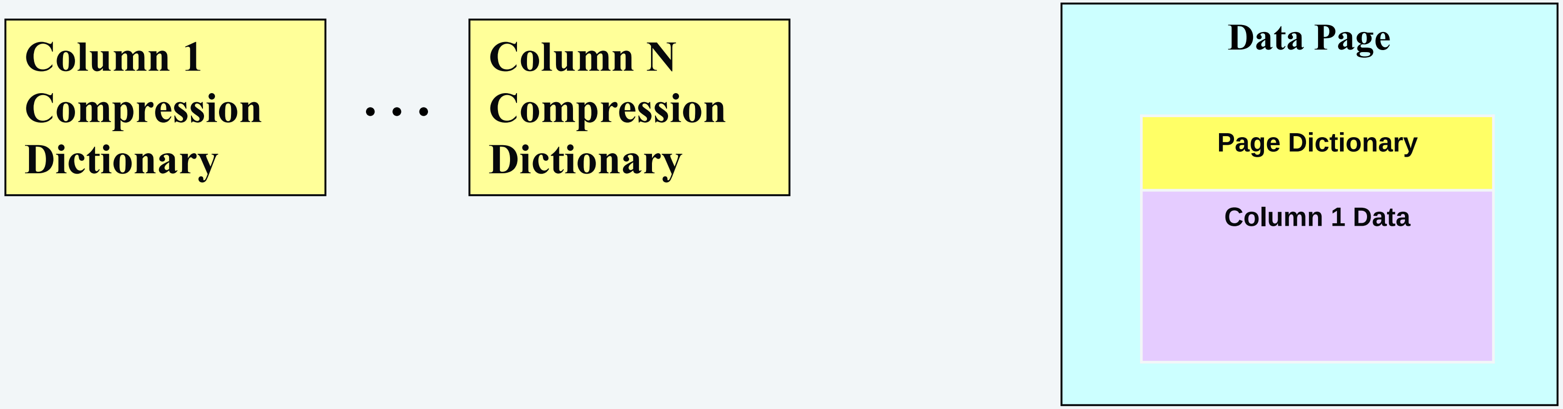
Actionable compression allows the following actions to be performed on compressed data

- Predicate evaluation (=, <, >, >=, <=, Between, LIKE)
- Group-by and Join processing on encoded data use global and On-The-Fly (OTF) encoding

Order-preserving encoding allows range predicates to be evaluated on compressed data

- Avoiding decompression provides significant query performance gains

Compression Dictionaries for Column-Organized Tables


Column 1 Compression Dictionary
. . .
Column N Compression Dictionary
Data Page
Page Dictionary
Column 1 Data

- Column-level dictionaries: **Always one per column**
  - Dictionary created during load replace, load insert, SQL insert/update

- Page-level dictionaries: **May also be created during load or insert**
  - Used if space savings outweighs cost of storing page-level dictionaries
  - Exploit local data clustering at page level

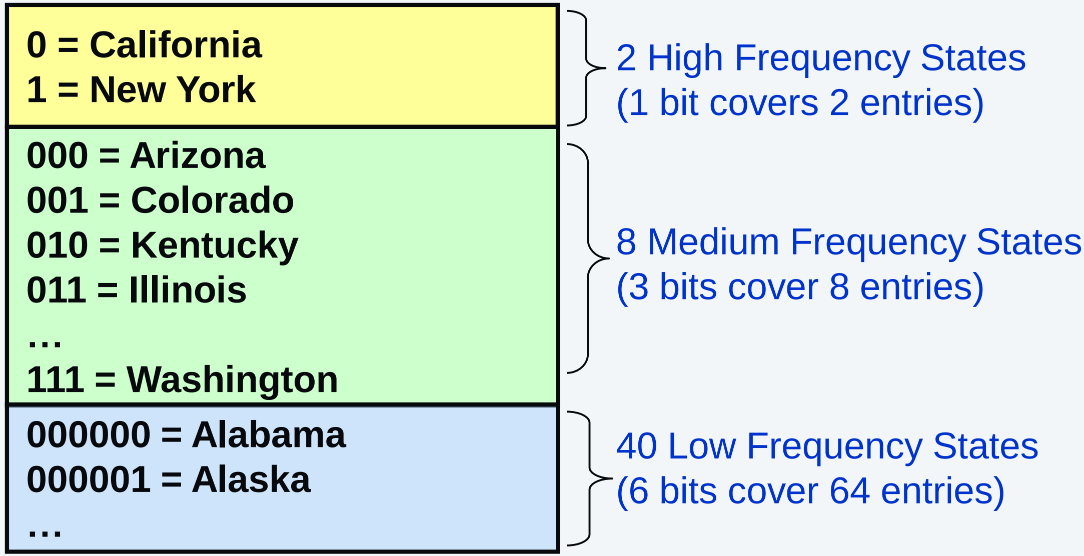
Columnar Compression 1 /2

Frequency (pure dictionary) encoding

- Most common values use fewest bits

Multiple compression techniques:

- Approximate Huffman-encoding
- Prefix encoding
- Offset encoding



0 = California
1 = New York

2 High Frequency States
(1 bit covers 2 entries)

000 = Arizona
001 = Colorado
010 = Kentucky
011 = Illinois
…
111 = Washington

8 Medium Frequency States
(3 bits cover 8 entries)

000000 = Alabama
000001 = Alaska
…

40 Low Frequency States
(6 bits cover 64 entries)

Columnar Compression 2/2

Prefix encoding

- Similar to approximate Huffman-encoding for common prefixes
- Prefix bits for encoded prefixes concatenated with uncompressed suffix bits
- Example values: MAR01, MAR02, JUN10, JUN15, etc.
- 4 prefix bits, 16 suffix bits

Offset (or minus) encoding

- Dictionary includes base value and number of bits that define range of coverage
- Example: Base value 0 and 10 offset bits (0-1023) , dates
- May include extended partition to provide some prediction for future values

Pure Dictionary Coding

NumDictBits = 3
NumOffsetBits = 0

Dictionary

Dictionary Index	Dictionary Values
0 = ‘000’b	Arizona
1 = ‘001’b	California
2 = ‘010’b	Florida
3 = ‘011’b	**Hawaii**

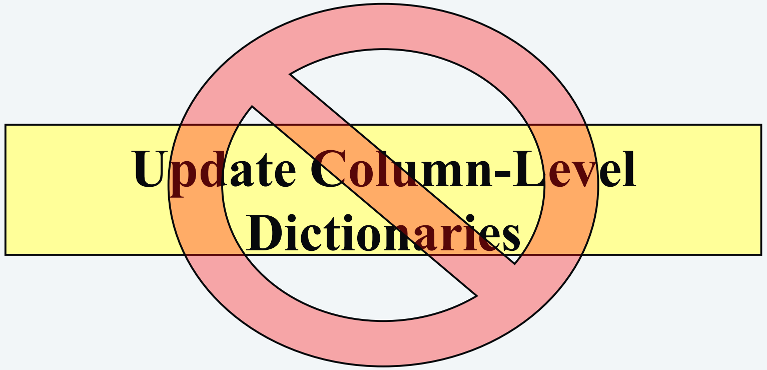
Encoded Value in Binary	DictionaryIndex	Offset	Decoded Value
011	3 = ‘011’b	0	011 >> 0 = 011 Decode (011) = dictValues[011] + 0 => **Hawaii**

Code = dictionary index alone with no offset bits

Set of base dictionary values with cardinality = 2 ** 3 => 8 possible dictionary values

In this example, 4 of the 8 possible dictionary values are shown

Column-Level Dictionaries are Static



Update Column-Level Dictionaries

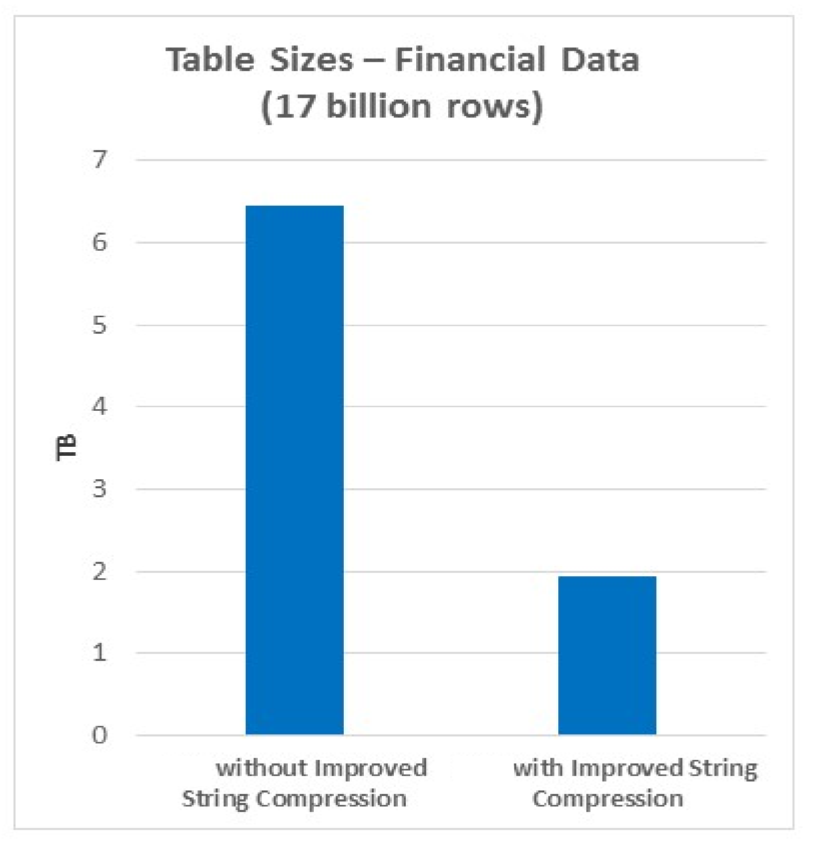
Page Compression

- Once created, evolved column-level dictionaries are static

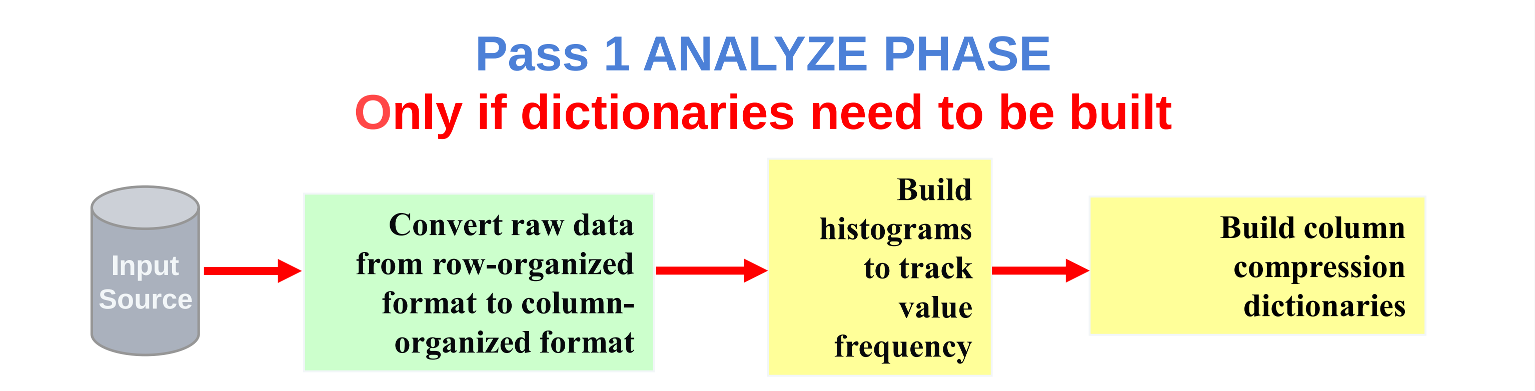
- Compression ratio may deteriorate if newly-inserted values are not covered by the column-level dictionary
    - Page compression can reduce need to rebuild column dictionaries
        - New offset encodeable values not covered by column-level dictionaries can still be compressed by page-level dictionaries

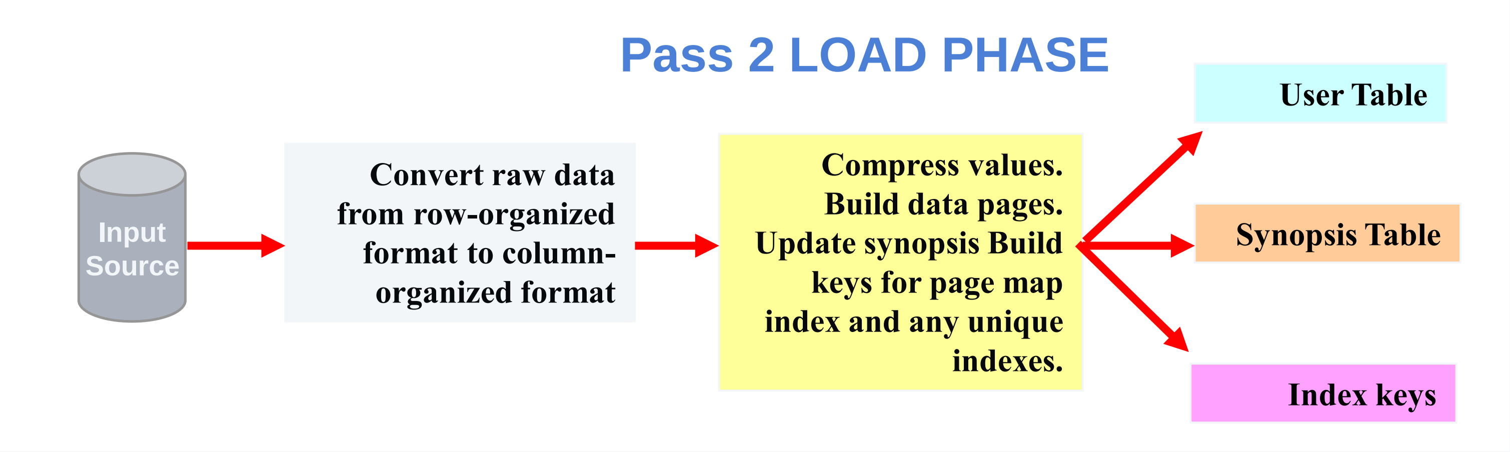
FUTURE: Improved Compression of String Datatypes

- Frequency-based compression difficult for some string datasets
- String data dominates storage cost
- Add another level of pattern-based page compression



Load for Column Organized Tables


Pass 1 ANALYZE PHASE
Only if dictionaries need to be built
Input Source
Convert raw data from row-organized format to column-organized format
Build histograms to track value frequency
Build column compression dictionaries


Pass 2 LOAD PHASE
Input Source
Convert raw data from row-organized format to column-organized format
Compress values. Build data pages. Update synopsis Build keys for page map index and any unique indexes.
User Table
Synopsis Table
Index keys

Maximizing Compression with the Load Utility

- Use sufficiently large amount of representative data in 1st Load that builds dictionaries
- Set util_heap_sz >= 1,000,000 pages with **AUTOMATIC option**
- Consider pre-sorting the input data by columns that are commonly referenced by predicates that filter the fact table or are often joined with dimension tables.
- To minimize amount of time table is offline and create a near-optimal dictionary
  - Step 1: Manually build dictionary using load utility and Bernoulli sampling
  - Step 2: Insert data
- Refer to Notes for an example

New in v11.5: External Tables

- **Query data in external files** (such as CSV text files) as though it were database data
- **Load from external files** through this interface
- **New data parser** – proven to parse > 16TB/hour

Example creating and querying an external table

```
create external table ext_orders(order_num INT, order_dt TIMESTAMP)
                USING(dataobject('/tmp/order.tbl') DELIMITER '|');

select COUNT(*) from ext_orders;
```

Example loading data from an external table

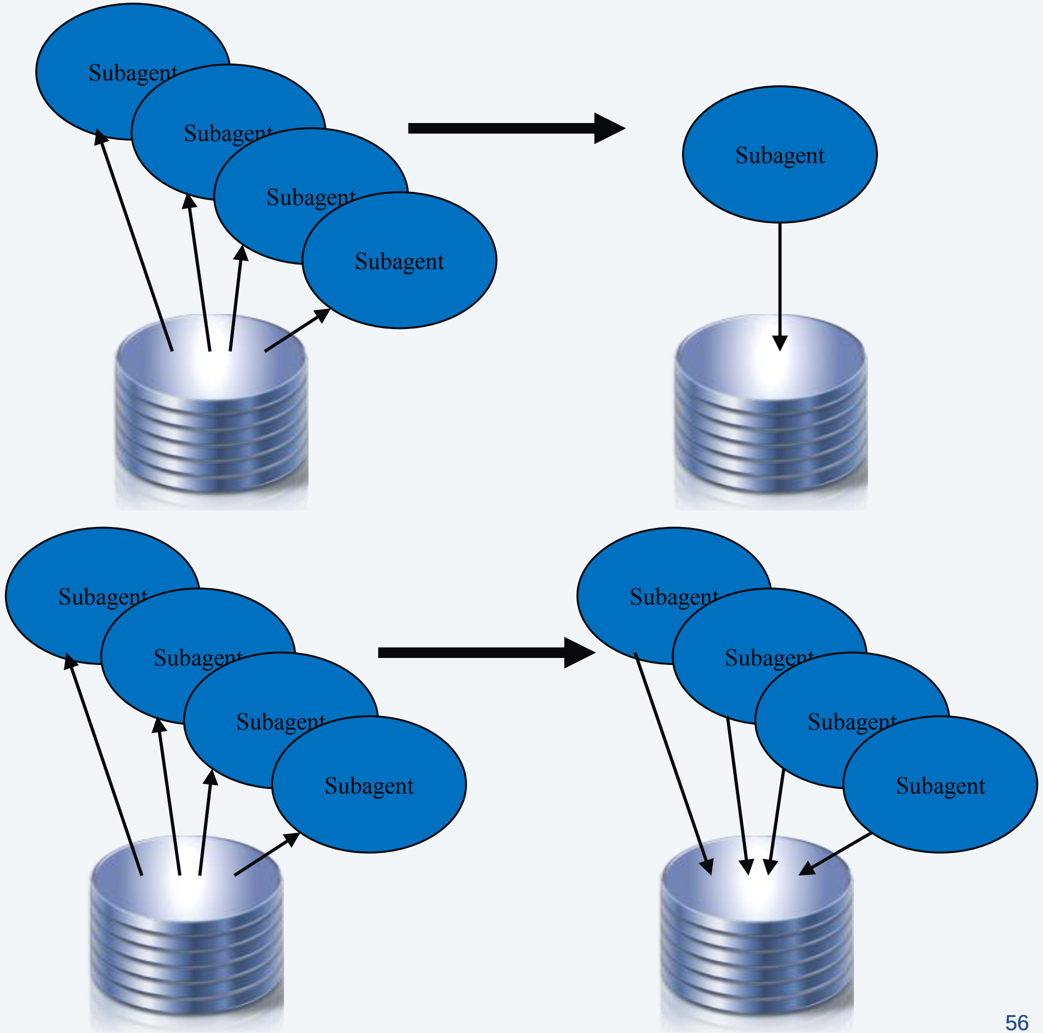
```
insert into orders select * from ext_orders;
```

New in v11.5: Insert/Update/Delete Performance Enhancements

- Db2 11.5 greatly expands core-friendly parallelism for SQL-based IUD operations on columnar tables
  - KIWI: Kill It With Iron
  - Maximize CPU cache, cache-line efficiency
- Critical to maximize ETL/ELT batch performance
- Many general improvements, but primary focus on bulk operations
  - See Notes for examples

Parallel Insert/Update/Delete

- BLU query processing leverages core-friendly parallelism
  - Excellent scalability for large SMPs
  - Combine SMP and MPP scaleout
- BLU bulk IUD now provides similar parallelism
  - Parallel insert available in v11.1.1.2
- INSERT INTO table2 SELECT * FROM table1

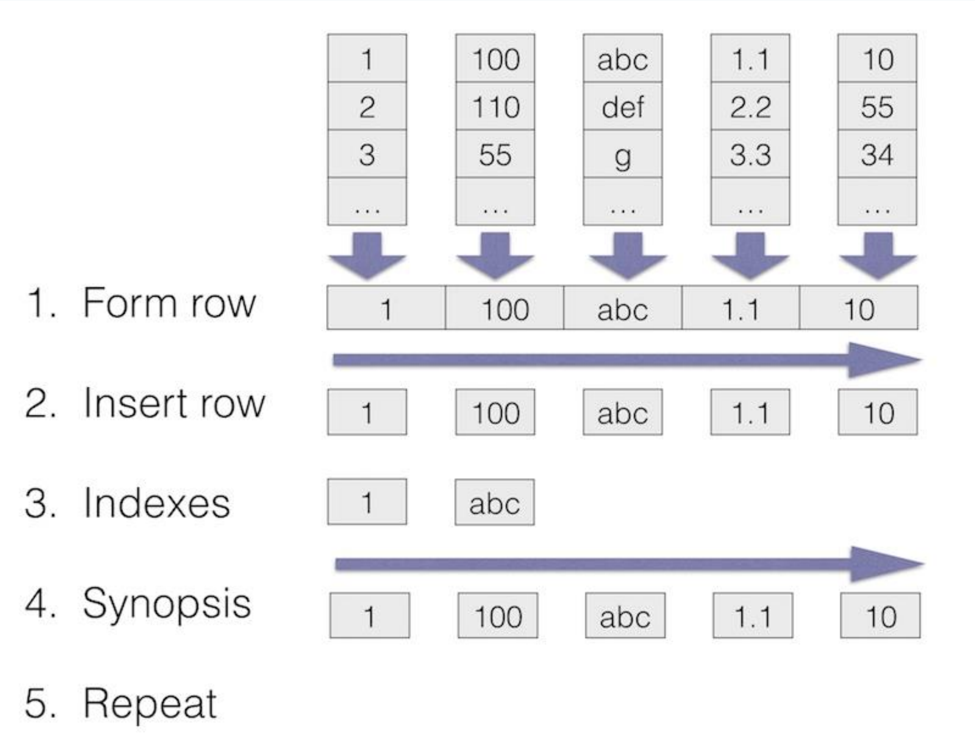


New in v11.5: Vectorized Insert/Update 1/2

- BLU query processing leverages vectors of columnar tuplets
  - Enables bulk processing on columns instead of row by row
  - Maximizes cache and cacheline efficiency
- Bulk insert/update operations benefit from similar access pattern

New in v11.5: Vectorized Insert/Update 2/2

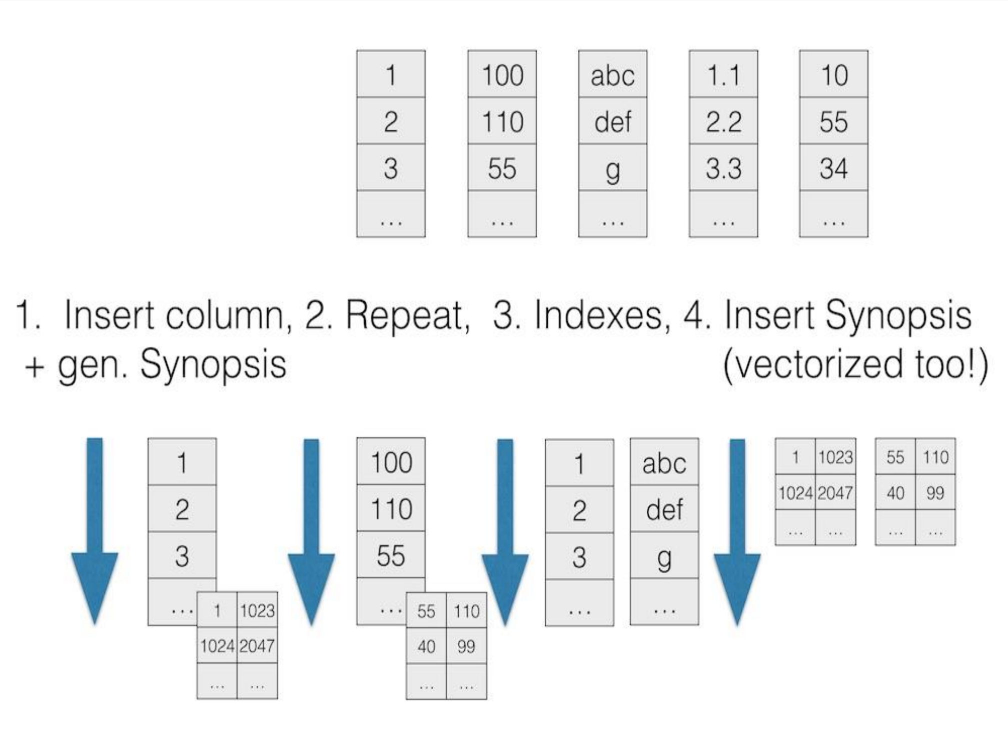
Pre-v11.5 approach



1	100	abc	1.1	10
2	110	def	2.2	55
3	55	g	3.3	34
…	…	…	…	…

1. Form row — 1 | 100 | abc | 1.1 | 10
2. Insert row — 1 | 100 | abc | 1.1 | 10
3. Indexes — 1 | abc
4. Synopsis — 1 | 100 | abc | 1.1 | 10
5. Repeat

v11.5 approach



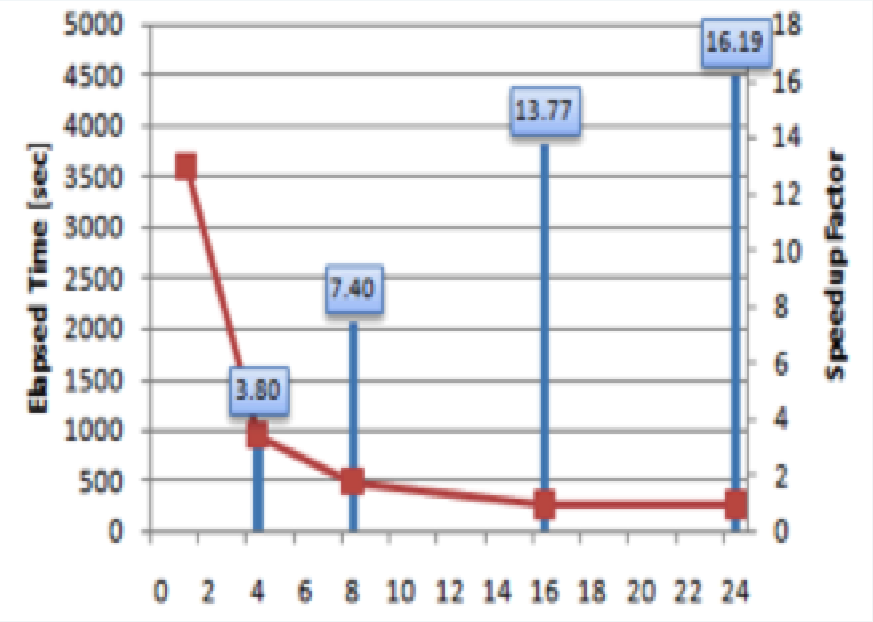
1	100	abc	1.1	10
2	110	def	2.2	55
3	55	g	3.3	34
…	…	…	…	…

1. Insert column, 2. Repeat, 3. Indexes, 4. Insert Synopsis
+ gen. Synopsis (vectorized too!)

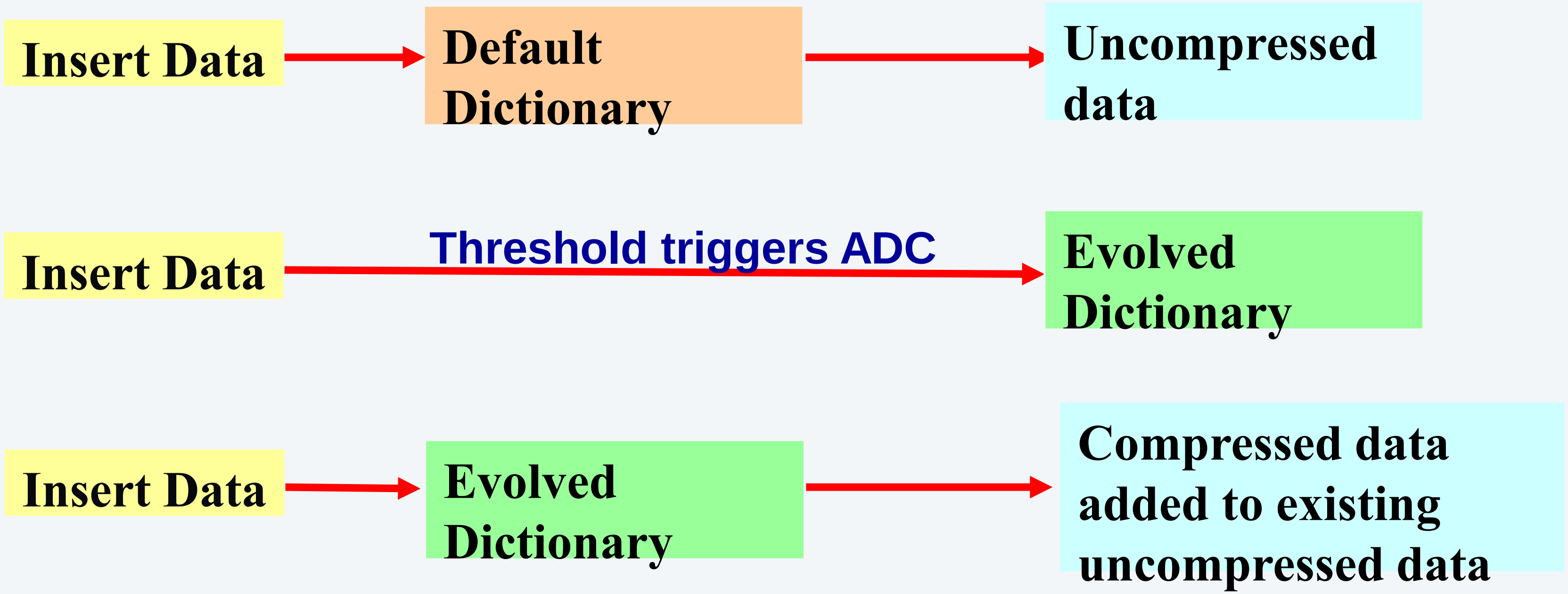
ETL Performance Example

- Data ingest rate
  - 1 TB/hour before enhancements
  - ***Now ~5 TB/hour (IIAS)***
- >10 TB data
- Table remains online
- Combined features
  - v11.5 -> ET load
  - v11.1 -> Parallel insert
  - v11.5 -> Vectorized insert
  - V11.5 -> Optimized bulk insert code path
  - Future -> reduced logging

Parallel Insert Degree/Time/Speedup



Automatic Dictionary Creation (ADC) during SQL Insert



Insert Data → Default Dictionary → Uncompressed data

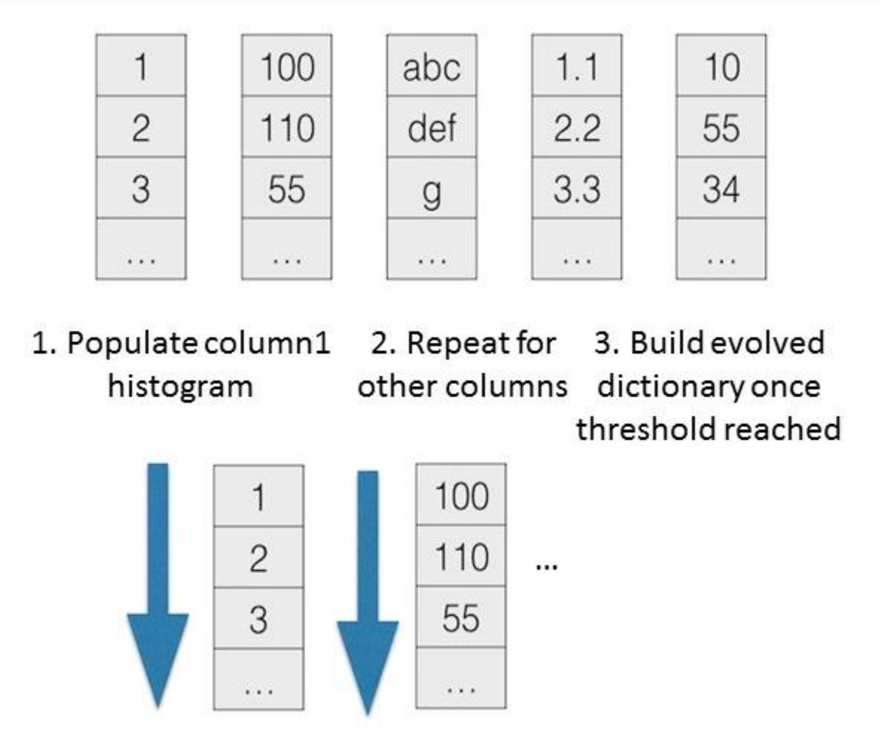
Insert Data → Threshold triggers ADC → Evolved Dictionary

Insert Data → Evolved Dictionary → Compressed data added to existing uncompressed data

- Initial data inserted before ADC is uncompressed
- Once ADC threshold is reached, ADC builds evolved dictionary
- 3 types of ADC: Vectorized, Synchronous, and Asynchronous

New in v11.5: Vectorized ADC

- Optimized for bulk insert
- Executes within insert threads, even across streams
- Maximizes cache and cacheline efficiency
- Once dictionary build starts, delay insert threads.



1. Populate column1 histogram

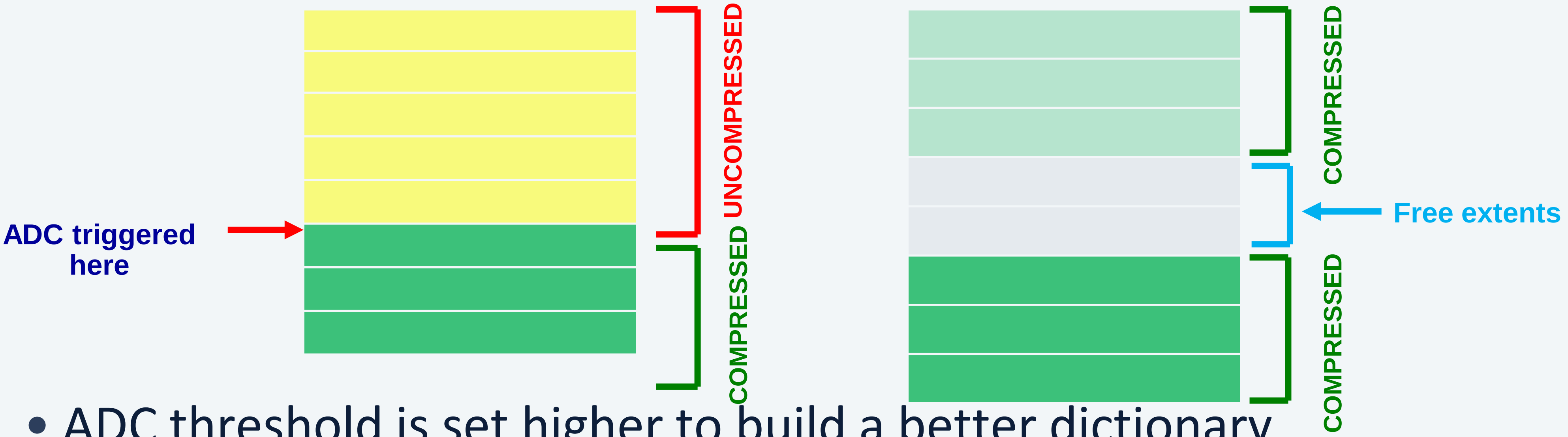
2. Repeat for other columns

3. Build evolved dictionary once threshold reached

Maximizing Compression with SQL Insert

- Bulk insert to empty table creates high-quality column-level dictionaries while minimizing time to creation
  - Should include sufficient number of rows to reach ADC threshold
  - Leverages vectorized ADC

New in v11.5: Automatic REORG Recompress



- ADC threshold is set higher to build a better dictionary
- Large number of values at the front of the table left uncompressed
- REORG Recompress automatically uses evolved dictionary to recompress committed data previously encoded using default dictionary
- Frees full extents, but does not deallocate them

Automatic Space Reclamation

After automatic REORG Recompress frees extents, a subsequent REORG TABLE…RECLAIM EXTENTS may return pages in freed extents to tablespace storage

These reclaimed pages may be reused by any tables in same tablespace

auto_reorg database configuration parameter controls if auto reclamation takes place

- Set to ON if DB2_WORKLOAD=ANALYTICS

Reducing Unused Space in a Tablespace

Once extents are reclaimed, they are available for reuse within the same tablespace

However, this unused space can also be released for other consumers

- A sample query to detect unused space is provided in the Notes

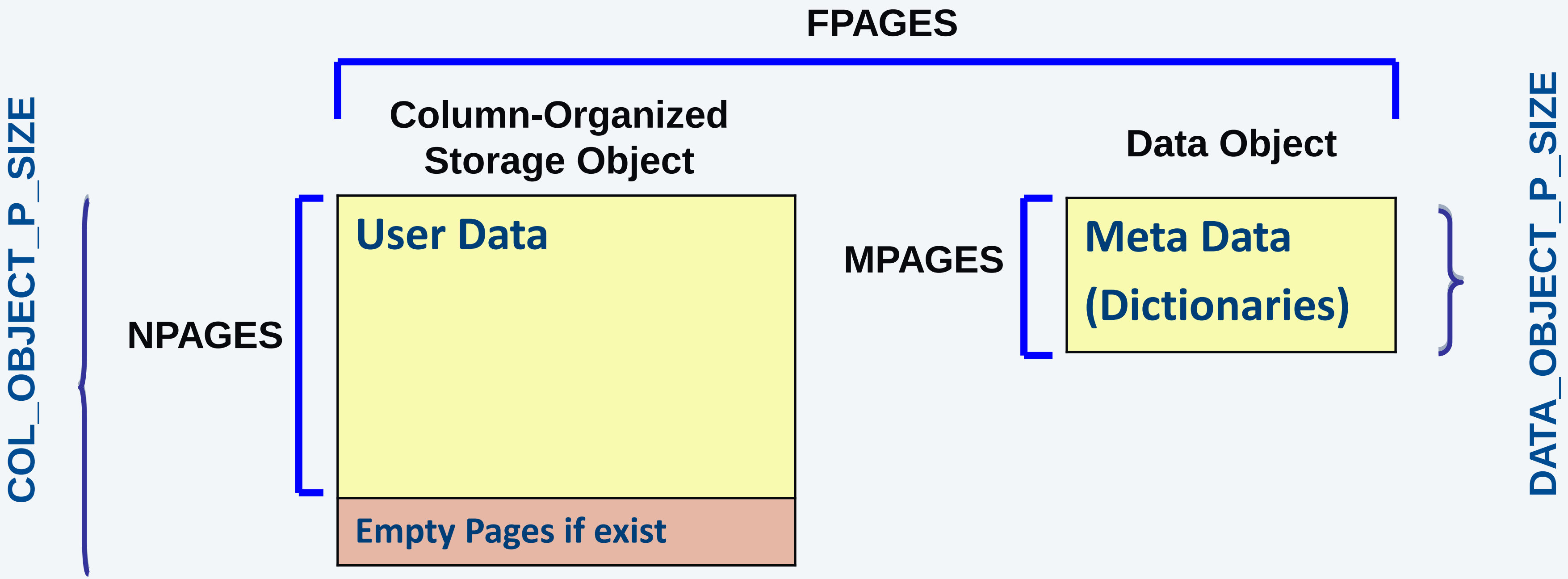
To release all unused space and lower the high water mark:

```
ALTER TABLESPACE <TBSPACE NAME> REDUCE MAX
```

Measuring Compression 1/2

- Statistics for Measuring Number of Pages in SYSCAT.TABLES
  - **NPAGES:** Number of pages in Column-Organized Object minus any empty pages
  - **FPAGES:** Total number of pages in both objects
  - **MPAGES:** (M for meta data) Number of pages in Data Object
- ADMIN_GET_TAB_INFO table function reports
  - **COL_OBJECT_P_SIZE:** Physical size (KB) of column data object containing user data
  - **DATA_OBJECT_P_SIZE:** Physical size (KB) of data object containing meta data

Measuring Compression 2/2



Calculating Column-Organized Storage Sizes

User Data	COL_OBJECT_P_SIZE
User Data + Meta Data + Page Map/Unique Indexes	COL_OBJECT_P_SIZE + DATA_OBJECT_P_SIZE + INDEX_OBJECT_P_SIZE

- Be careful using NPAGES to determine table size
  - May underestimate actual space usage especially for small tables
  - Doesn’t take meta data or empty pages into account
- Use the table function ADMIN_GET_TAB_INFO or admin view ADMINTABINFO to retrieve
  - COL_OBJECT_P_SIZE + DATA_OBJECT_P_SIZE + INDEX_OBJECT_P_SIZE

Table Compression Statistics in SYSCAT.TABLES

Row-Organized Table Statistics	Column-Organized Table Statistics
PCTPAGESSAVED	PCTPAGESSAVED
AVGCOMPRESSEDROWSIZE	
AVGROWCOMPRESSIONRATIO	
AVGROWSIZE	
PCTROWCOMPRESSED	

- Only PCTPAGESSAVED applies to column-organized tables too
  - Approximate percentage of pages saved in the table
  - Runstats collects PCTPAGESSAVED by estimating the number of data pages needed to store table in uncompressed row orientation
- ADMIN_GET_COMPRESS_INFO not supported yet for column-organized tables and will return zero rows

Estimating Compression Ratios

- PCTPAGESSAVED can be converted to a compression ratio
  - See Notes for sample query

Compression Ratio = Uncompressed Size / Compressed Size

= 1 / (1 - PCTPAGESSAVED/ 100)

PCTENCODED Statistic in SYSCAT.COLUMNS

✓

C1	PCTENCODED = 90
C2	PCTENCODED = 75
C3	PCTENCODED = 100

⚠

C1	PCTENCODED = 0
C2	PCTENCODED = 10
C3	PCTENCODED = 0

- Monitor this statistic to determine how many values were left uncompressed in specific columns
- Percentage of values encoded (compressed) by column-level dictionary
- It measures number of values compressed NOT compression ratio

PCTENCODED Example

COLUMN	TYPENAME	LENGTH	CARD	AVGCOLLEN	AVG_ENCODED_LEN	COMP_RATIO	PCTENCODED
Prod_Info	VARCHAR	40	6114112	20	17.48	1.14	30
Comment	VARCHAR	600	4022272	141	139	1.01	10
Code	VARCHAR	3	4	5	0.17	28.21	98
Cust_Num	VARCHAR	80	8145280	12	5.00	3.19	99

Sample values from Cust_Num shows common prefixes:
`0000002807200, 0000002807211, 0000002807222, 0000002807233`

- CHAR and VARCHAR values with high cardinality and no common prefix do not compress well until Text Compression feature delivered
- Prod_Info and Comment have high cardinality and no common prefix
- Code has low cardinality
- Cust_Num has high cardinality and common prefixes

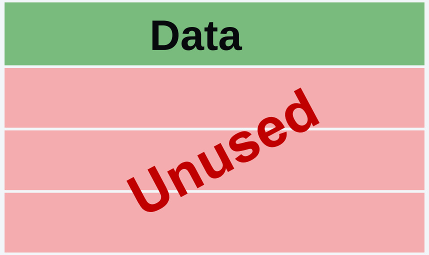
Maximizing Compression with Data Skew 1/2

- Over time, column-level dictionaries may become less representative of a table’s data
  - PCTENCODED decreases
  - Page compression may help maintain an acceptable compression ratio even with new values
  - If PCTENCODED decreases especially for columns used for joining or grouping, query performance impact is possible

Maximizing Compression with Data Skew 2/2

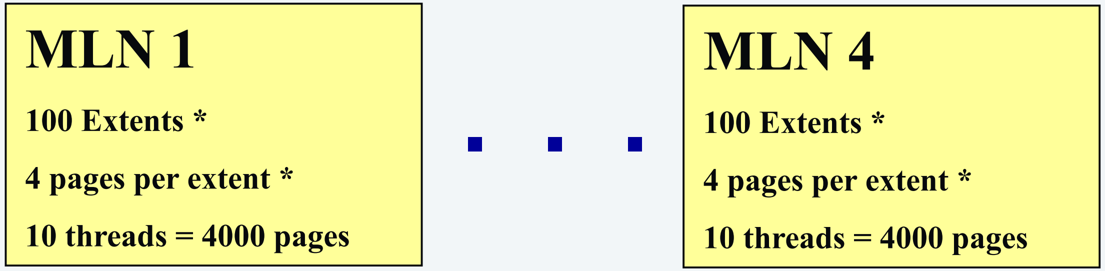
- It is recommended to monitor PCTENCODED values over time for such tables/columns that have frequent insert/update/delete activity.
- If you notice that PCTENCODED values are dropping notably lower and query performance is important for the column:
  - Option 1: Unload and reload the table including rebuilding the column-level dictionaries
  - Option 2: Create a new empty table, use the load utility to build a high quality dictionary, insert data into the new table, drop the old table, and rename the new table
  - Option 3: Use ADMIN_MOVE_TABLE to update your table
- See Notes for more info on these options

Space Usage Overhead for Small/Medium Tables



Default tablespace extent size 4

with only 1 page out of 4 used



MLN 1

100 Extents *

4 pages per extent *

10 threads = 4000 pages

. . .

MLN 4

100 Extents *

4 pages per extent *

10 threads = 4000 pages

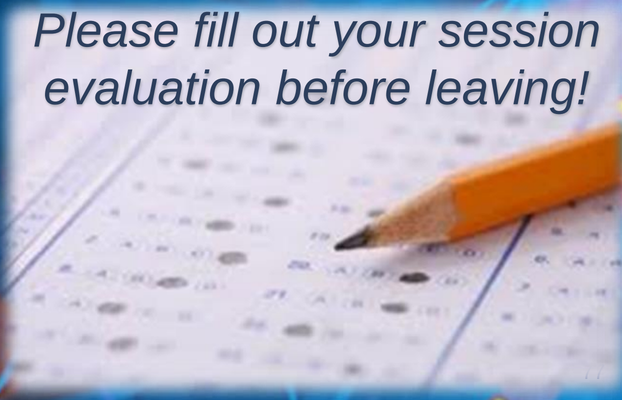
Table with 100 columns using default of 4 pages per extent and Parallel Degree 10 on MPP system with 4 MLNs = 16000 allocated pages with only 4000 used pages

- 1 column per extent
- Small and medium tables may only use 1 page per extent which leaves rest of pages in extent unused but available for more data
- Changing tablespace extent size to 2 reduces overhead for small and medium tables

Keri Romanufa
IBM
keri@ca.ibm.com

Session code: TRIDEX

IDUG
Leading the Db2 User
Community since 1988



Please fill out your session evaluation before leaving!

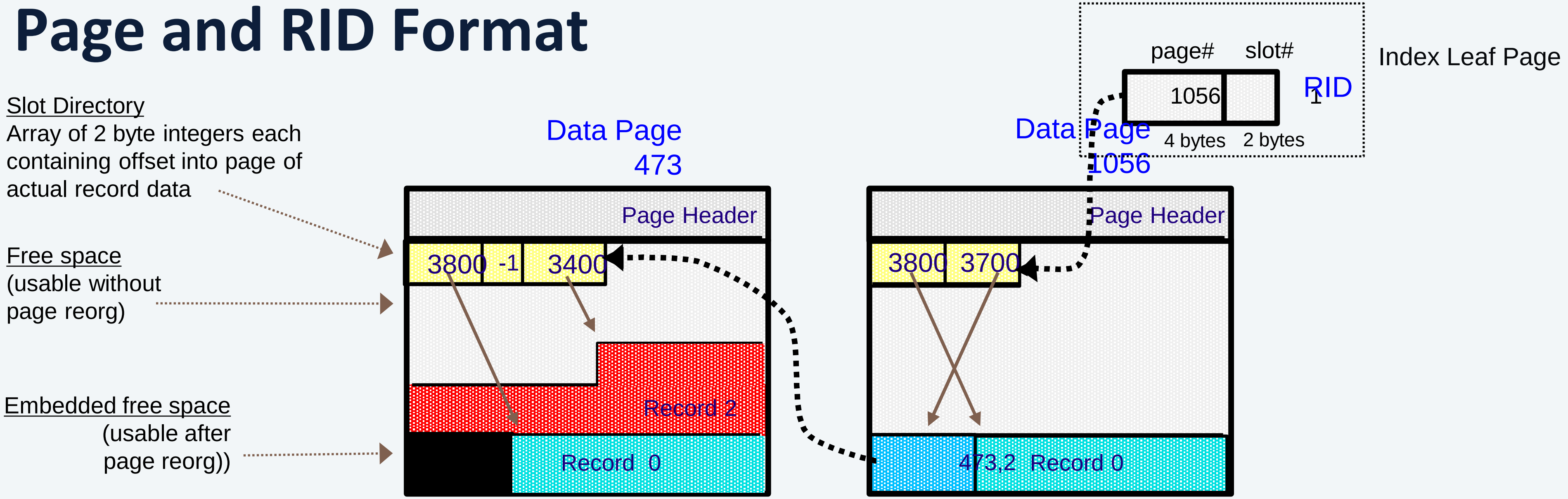
Bonus Material for Row based

Some Bonus Material on Row based PART 1b:

- Architecture Overview
    - Basic Operation Walkthroughs

- Table Management
    - Tables, Records
    - Page Format, Space Management
    - Row Compression (including Adaptive Compression)
    - Currently Committed
    - IUD Logging
    - Space Mmgt & Clustering
        - FSCR Search
        - Append Mode
        - Clustered Index
        - Multi-Dimensional Clustering
        - Insert Time Clustered Tables
    - Range Partitioned Tables
    - Indexes
    - Columnar (aka BLU) Tables & Compression

Data Page and RID Format



Slot Directory
Array of 2 byte integers each containing offset into page of actual record data

Free space
(usable without page reorg)

Embedded free space
(usable after page reorg))

Notes

1) Page reorgs are done automatically online as required. They can be monitored via MON_GET_TABLE()

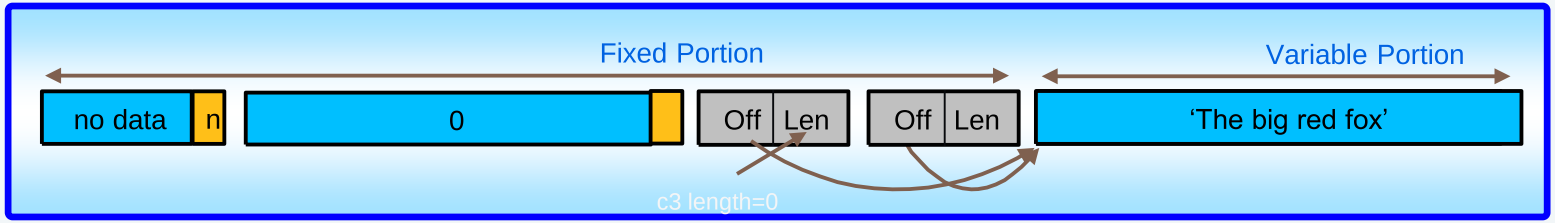
2) Free space created by deletes or updates can be held reserved (not usable) until the delete transaction is committed and older than:
a) the oldest transaction reading the table, or,
b) the oldest modifying transaction in the db

Tip Use larger page sizes for workloads that tend to access rows sequentially (eg. Warehousing, TEMP tables) and smaller page sizes for random access workloads (eg OLTP)

Tip If deleted space is not being reused,…
…. look for long-running transactions
(eg. APPLID_HOLDING_OLDEST_XACT from MON_GET_TRANSACTION_LOG())

Default Row Format

```
CREATE TABLE t1 (  c1 INTEGER,
                   c2 DECIMAL(12),
                   c3 VARCHAR(20) NOT NULL,
                   c4 VARCHAR(50) NOT NULL  )

INSERT INTO T1 VALUES ( null ,0,'','The big red fox')
```



Legend:

🟦 Actual column data (n bytes)

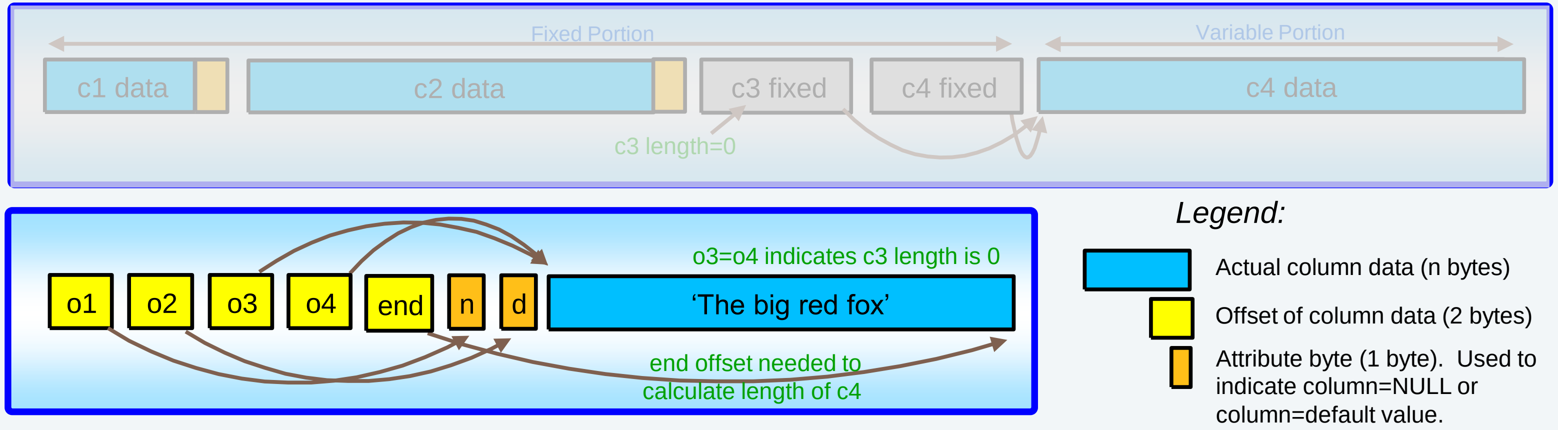
Off | Len Fixed portion of variable column (4 bytes : offset + length)

🟨 Attribute byte (1 byte)

- is only present for NULLable columns
- indicates if the value is in fact NULL

Alternate Row Format & Value Compression

```
CREATE TABLE t1 (  c1 INTEGER       COMPRESS SYSTEM DEFAULT,
                   c2 DECIMAL(12) COMPRESS SYSTEM DEFAULT,
                   c3 VARCHAR(20) NOT NULL,
                   c4 VARCHAR(50) NOT NULL  )
                 VALUE COMPRESSION

INSERT INTO T1 VALUES (null,0, '', 'The big red fox')
```


Fixed Portion
Variable Portion
c1 data
c2 data
c3 fixed
c4 fixed
c4 data
c3 length=0
o3=o4 indicates c3 length is 0
o1
o2
o3
o4
end
n
d
'The big red fox'
end offset needed to calculate length of c4
Legend:
Actual column data (n bytes)
Offset of column data (2 bytes)
Attribute byte (1 byte). Used to indicate column=NULL or column=default value.

Tip Consider alternate row format (VALUE COMPRESSION keyword) when …

- Significant # of rows contain the column default values (eg. 0 for numerics)
- Significant # of rows contain NULL column values
- Significant # of variable length columns

Extended Row Size

Page size	Row size limit	# Col limit
4 K	4 005	500
8 K	8 101	1 012
16 K	16 293	1 012
32 K	32 677	1 012

Allows a table to be larger than the page size maximum.

```
CREATE TABLE t1 (  c1 INTEGER,    c2 DECIMAL(12),
                   c3 VARCHAR(3000) NOT NULL,   c4 VARCHAR(3000)  NULL,
                   c5 VARCHAR(3000) NOT NULL  )                    // in a 4k page tbsp

INSERT INTO T1 VALUES (null, 0, ‘Hello’, null, ’World’)
    → whole row is in the data page

INSERT INTO T1 VALUES (null, 0, ‘Hello’, repeat(‘x’,2500), repeat(‘x’,2500))
    → overflow  (1 varchar column)  is replaced by a 24byte descriptor and the data is moved into a large object (LOB)
    → varchars <-24 are never replaced
```

Requirements for enabling extended row size support for a table:

- The extended_row_sz database configuration parameter must be set to ENABLE. (default for new DB’s)
- The table definition must contain at least one varying length string column (VARCHAR or VARGRAPHIC).
- The row size of the table cannot exceed 1048319 bytes (SQLSTATE 54010).
- Queries requiring an explicit or implicit system temporary table with extended rows needs a system temporary table space that can fully contain the minimum width of the row. The minimum width of the row is calculated in the same way as the maximum width except that all VARCHAR and VARGRAPHIC columns are assumed to have a length of 1.

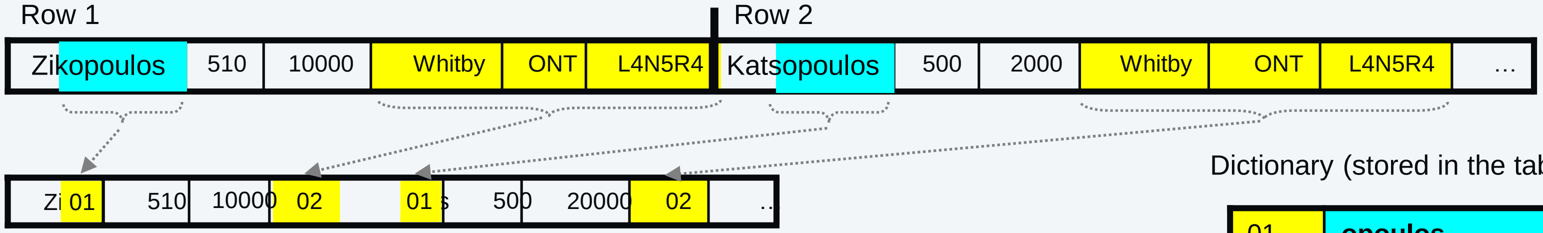
(Table-dict) Row Compression

Rows compressed in buffer pool, disk, logs, backup images

Dictionary-based LZ compression replaces frequently used byte sequences with 12-bit symbol

- Byte sequences can span column boundaries or within columns
- Global view of symbol frequency (not limited to single page)

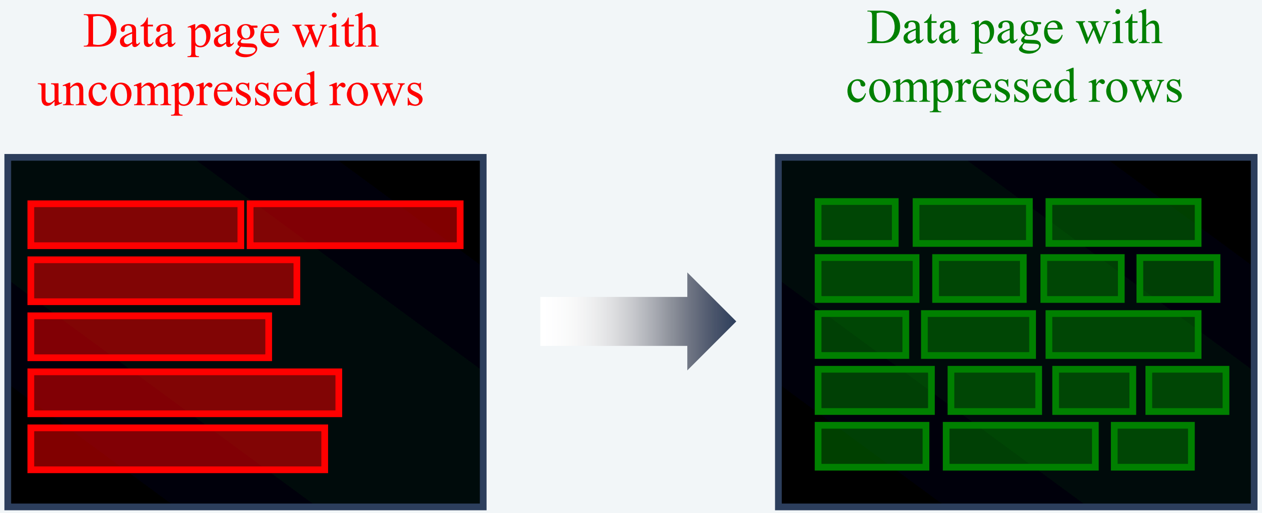
Name	Dept	Salary	City	Province	Postal_Code
Zik**opoulos**	510	10000	**Whitby**	**ONT**	**L4N5R4**
Kats**opoulos**	500	20000	**Whitby**	**ONT**	**L4N5R4**



Dictionary (stored in the table)

01	**opoulos**
02	**WhitbyONTL4N5R4**
…	…

(Table-dict) Row Compression


Data page with uncompressed rows
Data page with compressed rows

Effective in buffer pool and on-disk

- ❑ Saves memory
- ❑ Saves storage

Db2 Adaptive Compression : Overview

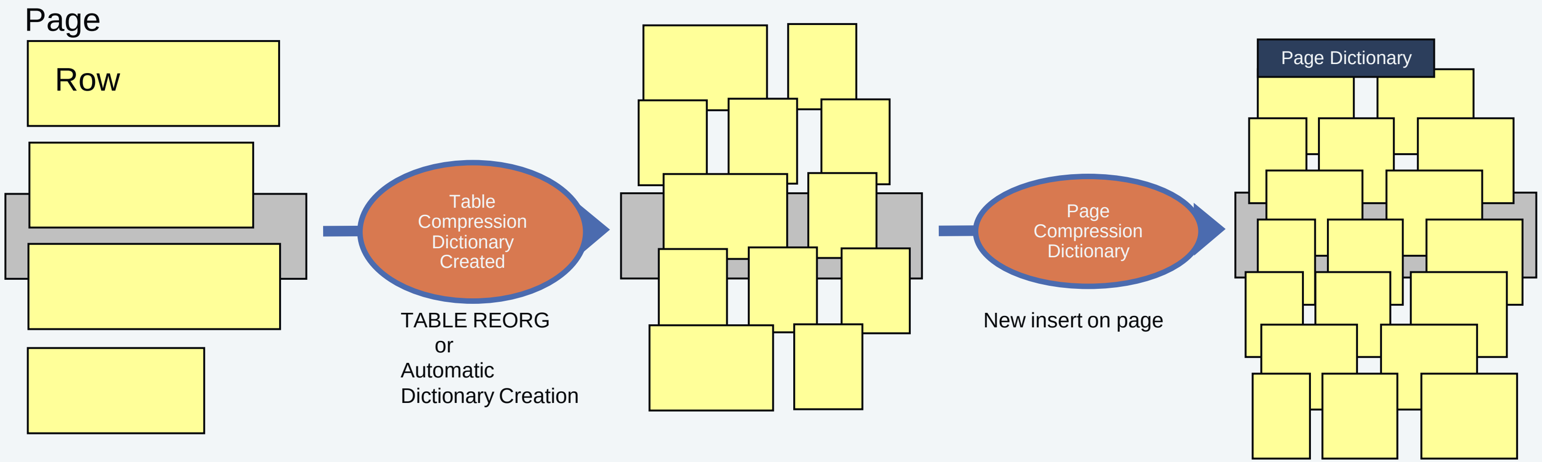
Db2 v10- *added* a *page level dictionary* to *further* compress page common symbols

- Adapts to changing data patterns

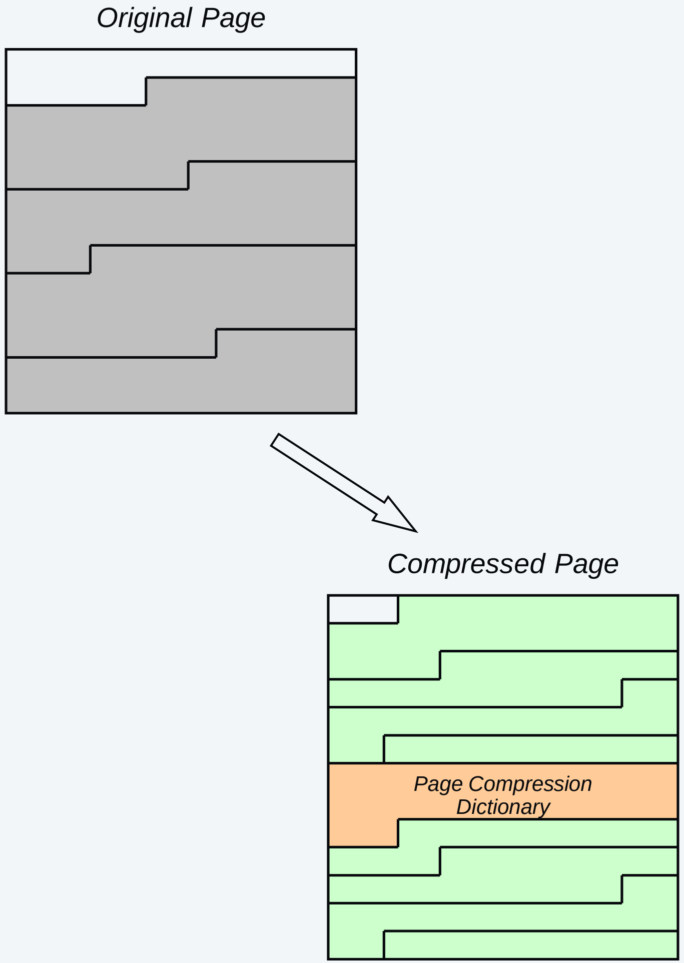
New keywords on ALTER/CREATE TABLE .. COMPRESS

- ADAPTIVE (default)
- STATIC

```
ALTER TABLE … COMPRESS YES
ALTER TABLE … COMPRESS YES ADAPTIVE
ALTER TABLE … COMPRESS YES STATIC
```



Adaptive Compression : How it Works



1. Rows are inserted into a page (compressed via table dictionary)
2. When page is almost full, page dictionary is built
3. Detect common recurring patterns in original records
4. Build compressed page by compressing all existing records
5. Insert page compression dictionary (special record)
6. Insert more compressed records in additional free space

Compression : Hints / Tips / Reminders

Group correlated columns together in table definitions

- E.g. place ‘Make’ (eg. Honda) and ‘Model’ (eg. Accord) columns adjacent to each other
- Db2’s row compression will compress common byte sequences regardless of column boundaries

If you created table spaces prior to V9.1, ensure you’ve enabled Large RIDs and Large Slots if more than 255 compressed rows will typically fit on your data pages

- Otherwise, Db2 will only place a maximum of 255 rows per page, resulting in less efficient utilization of memory and storage
- Call ADMIN_GET_TAB_INFO() and check LARGE_SLOTS and LARGE_RIDS and for ‘Y’

When using adaptive compression remember …

- New tables:
  - COMPRESS YES defaults to ADAPTIVE
  - (Can explicitly specify COMPRESS YES STATIC or COMPRESS YES ADAPTIVE)
- Pre-10.1 tables:
  - By default, will stay with existing (static) compression
  - Use ALTER TABLE … COMPRESS YES ADAPTIVE to enable adaptive compression dynamically

Estimate compression savings with ADMIN_GET_TAB_COMPRESS_INFO()

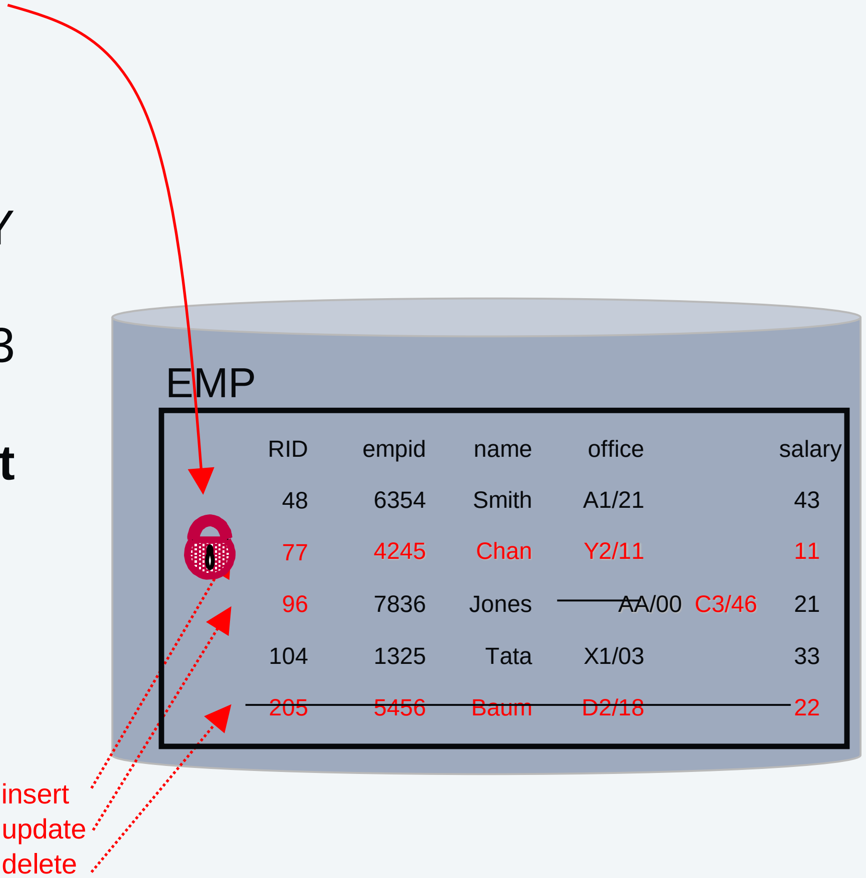
Report actual compression rate with ADMIN_GET_TAB_DICTIONARY_INFO()

Currently Committed Isolation : Motivation

SELECT * FROM EMP

EMPID	NAME	OFFICE	SALARY
6354	Smith	A1/21	43

> wait



EMP

RID	empid	name	office	salary
48	6354	Smith	A1/21	43
77	4245	Chan	Y2/11	11
96	7836	Jones	~~AA/00~~ C3/46	21
104	1325	Tata	X1/03	33
~~205~~	~~5456~~	~~Baum~~	~~D2/18~~	~~22~~

Uncommitted insert
Uncommitted update
Uncommitted delete

Currently Committed Isolation : Result

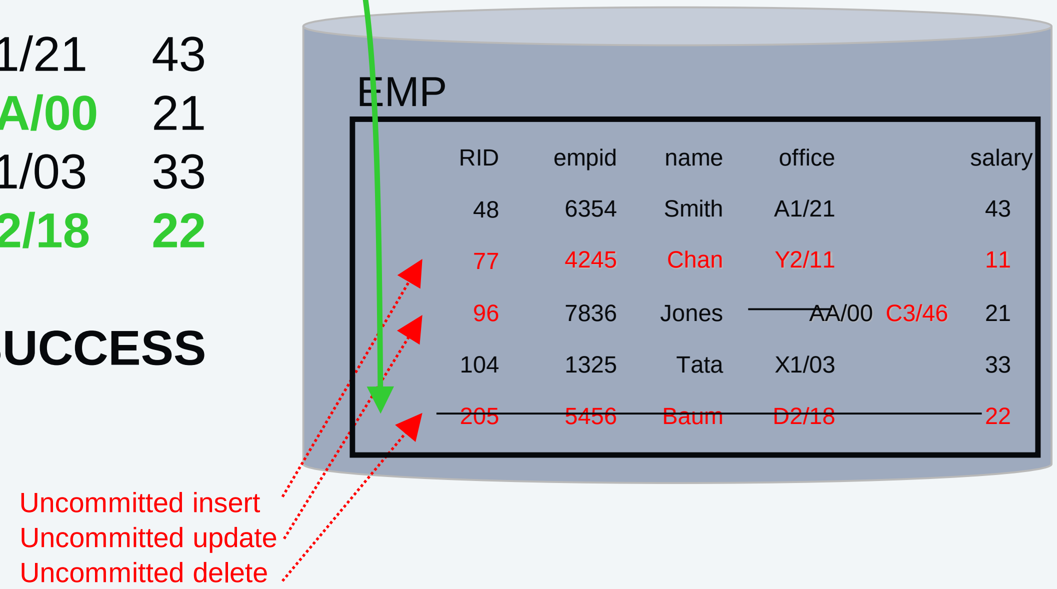
SELECT * FROM EMP

EMPID	NAME	OFFICE	SALARY
6354	Smith	A1/21	43
7836	Jones	**AA/00**	21
1325	Tata	X1/03	33
5456	**Baum**	**D2/18**	**22**

> SUCCESS

Db2 returns currently committed data without waiting for locks !

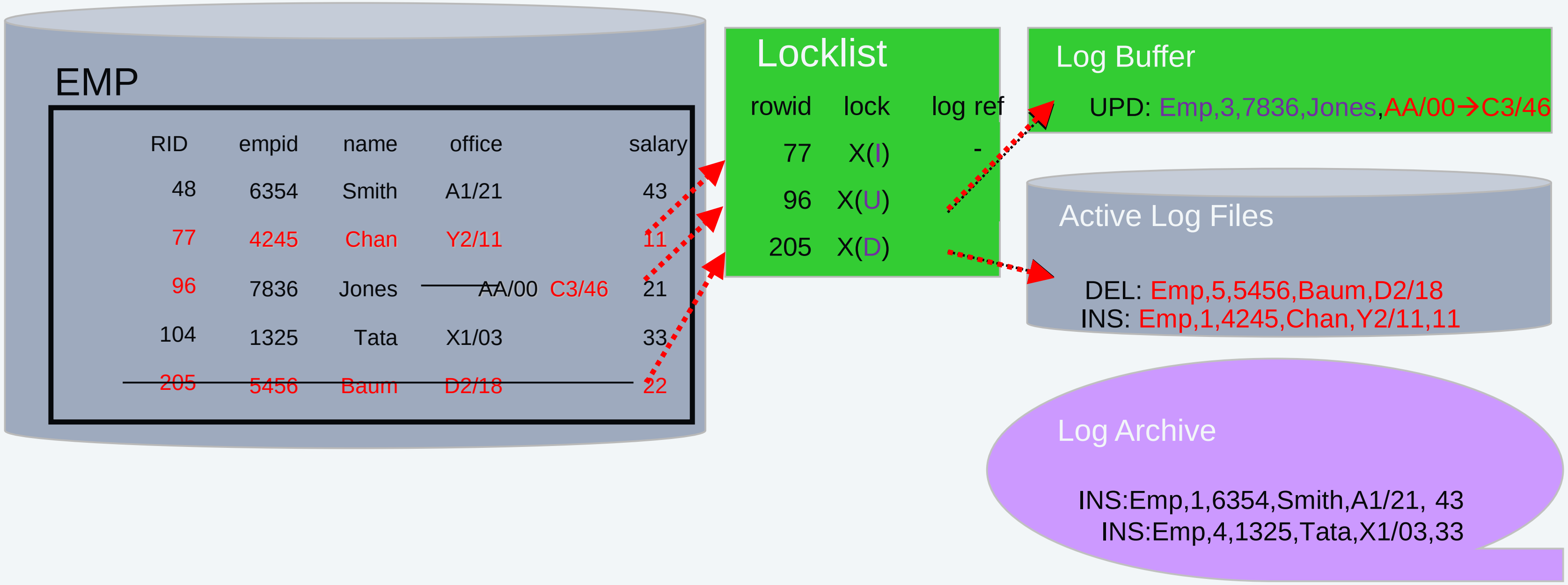
(Delete and Update undone; Insert skipped.)



EMP

RID	empid	name	office	salary
48	6354	Smith	A1/21	43
77	4245	Chan	Y2/11	11
96	7836	Jones	~~AA/00~~ C3/46	21
104	1325	Tata	X1/03	33
~~205~~	~~5456~~	~~Baum~~	~~D2/18~~	22

Currently Committed : How Does it Work ?



EMP

RID	empid	name	office	salary
48	6354	Smith	A1/21	43
77	4245	Chan	Y2/11	11
96	7836	Jones	~~AA/00~~ C3/46	21
104	1325	Tata	X1/03	33
~~205~~	~~5456~~	~~Baum~~	~~D2/18~~	22

Locklist

rowid	lock	log ref
77	X(I)	-
96	X(U)	
205	X(D)	

Log Buffer

UPD: Emp,3,7836,Jones,AA/00→C3/46

Active Log Files

DEL: Emp,5,5456,Baum,D2/18
INS: Emp,1,4245,Chan,Y2/11,11

Log Archive

INS:Emp,1,6354,Smith,A1/21, 43
INS:Emp,4,1325,Tata,X1/03,33

Uncommitted INSERTed data is skipped.

For uncommitted DELETEs and UPDATEs, when encountering a lock which would otherwise conflict, Db2 uses new information in the lock manager to reconstruct and return the previously committed data from the log buffer or log file.

As of 11.5 now works cross-member in pS

Currently Committed : Internals & Usage Notes

Log-based implementation : simple & fast

- No need for rollback segments
- Currently committed data typically reconstructed from memory (log buffer)
  - Exception: updates/deletes from mass update transactions that spill log buffer (active logs read from storage in this case)

Fallback to traditional locking

- If the currently committed data is unavailable (or not available quickly), Db2 will fall back to the traditional locking behavior
- Examples
  - Currently committed data is only available from an archived log (as may be the case with infinite logging)
  - Updater held table lock (not row lock)

Usage hints & tips

- Consider increasing your log buffer size if you see increased log disk reads
  - Use MON_GET_TRANSACTION_LOG() to check:
    CUR_COMMIT_DISK_LOG_READS - ideally want this close to 0
- Consider increasing lock list size (or using AUTOMATIC setting)
  - To avoid escalation to table locks (disables currently committed behavior for the table)
- Be aware of potential for small increase in log space consumption if CC enabled
  - First update to a given row in a transaction logs entire row image

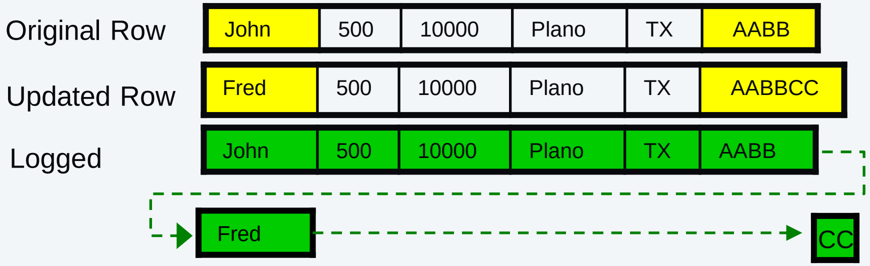
Sidebar: Row Logging

SQL	What's Logged
INSERT	RID + New row image
DELETE	RID + Old row image
UPDATE	RID + Four different cases,…

UPDATE Row Logging

What’s logged:

Full before image, plus after image of **changing** bytes and **new** bytes (if row is growing).



Original Row	John	500	10000	Plano	TX	AABB
Updated Row	Fred	500	10000	Plano	TX	AABBCC
Logged	John	500	10000	Plano	TX	AABB
	Fred					CC

When used:

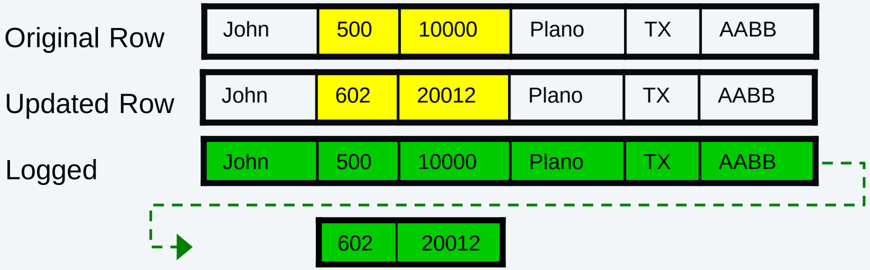
1. Currently Committed is enabled,

   - and -

2. First update to a given row in a given transaction,

   - and -

3. DATA CAPTURE CHANGES not in effect.



Original Row	John	500	10000	Plano	TX	AABB
Updated Row	John	602	20012	Plano	TX	AABB
Logged	John	500	10000	Plano	TX	AABB
		602	20012			

UPDATE Row Logging

What’s logged:

XOR between old and new rows
from 1st changed column to last changed column.

When used:

1. Currently Committed not in effect, (or, CC is in effect and transaction is updating given row again),

   - and -

2. Row length is not changing,

   - and -

3. DATA CAPTURE CHANGES not in effect.

Original Row	John	500	10000	Plano	TX	24357
Updated Row	Fred	500	10000	Plano	TX	24355
Logged	‘1A35D8C9E88719A6C23340037DCEFF8928D0A7883’x					

Original Row	John	500	10000	Plano	TX	AABB
Updated Row	John	602	20012	Plano	TX	AABB
Logged		‘18A0FF33C’x				

Tip

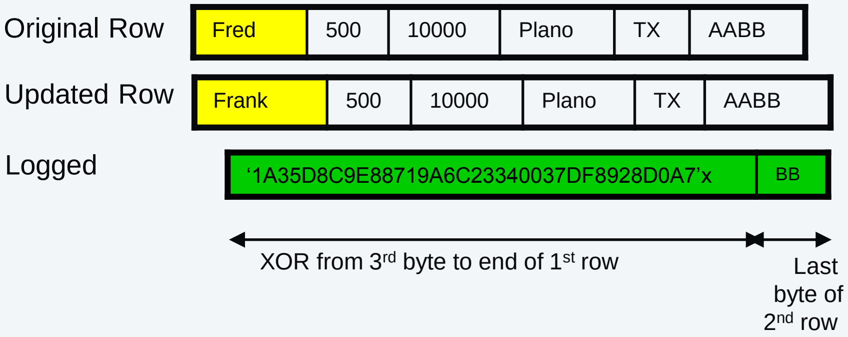
When UPDATES comprise a significant portion of your workload …

- Weigh extra UPDATE logging vs concurrency benefits of currently committed
- Try to place frequently updated columns adjacent in row definition

UPDATE Row Logging

What’s logged:

XOR between new & old row from 1st word that changes to end of smaller row version; then any residual words from larger row version.



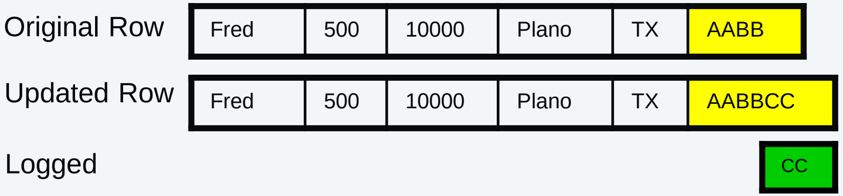
Original Row	Fred	500	10000	Plano	TX	AABB
Updated Row	Frank	500	10000	Plano	TX	AABB
Logged	‘1A35D8C9E88719A6C23340037DF8928D0A7’x					BB

XOR from 3rd byte to end of 1st row

Last byte of 2nd row

When used:

Same scenario as previous except row length is changing.



Original Row	Fred	500	10000	Plano	TX	AABB
Updated Row	Fred	500	10000	Plano	TX	AABBCC
Logged						CC

Tip

When UPDATES comprise a significant portion of your workload …

- Try to place frequently updated columns at end of row definition

UPDATE Row Logging

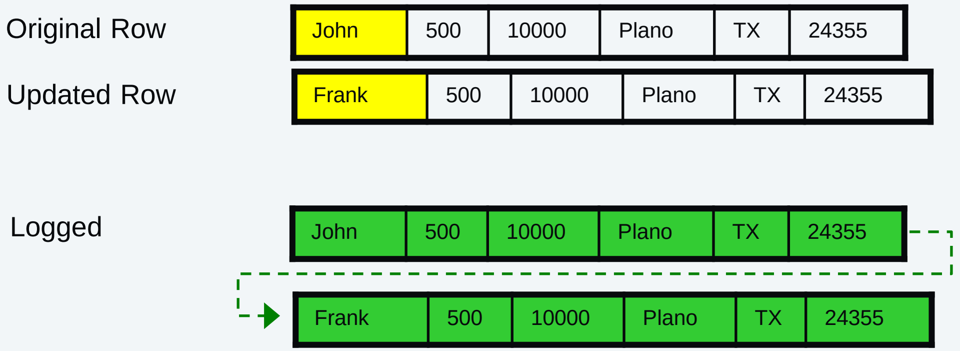
“Full Before & After Row Image”

What’s logged:

Full copies of old and new rows

When used:

Whenever DATA CAPTURE CHANGES (replication) is in effect for the table.



Original Row	John	500	10000	Plano	TX	24355
Updated Row	Frank	500	10000	Plano	TX	24355
Logged	John	500	10000	Plano	TX	24355
	Frank	500	10000	Plano	TX	24355

INSERT Processing (& Space Mgt) 1/2

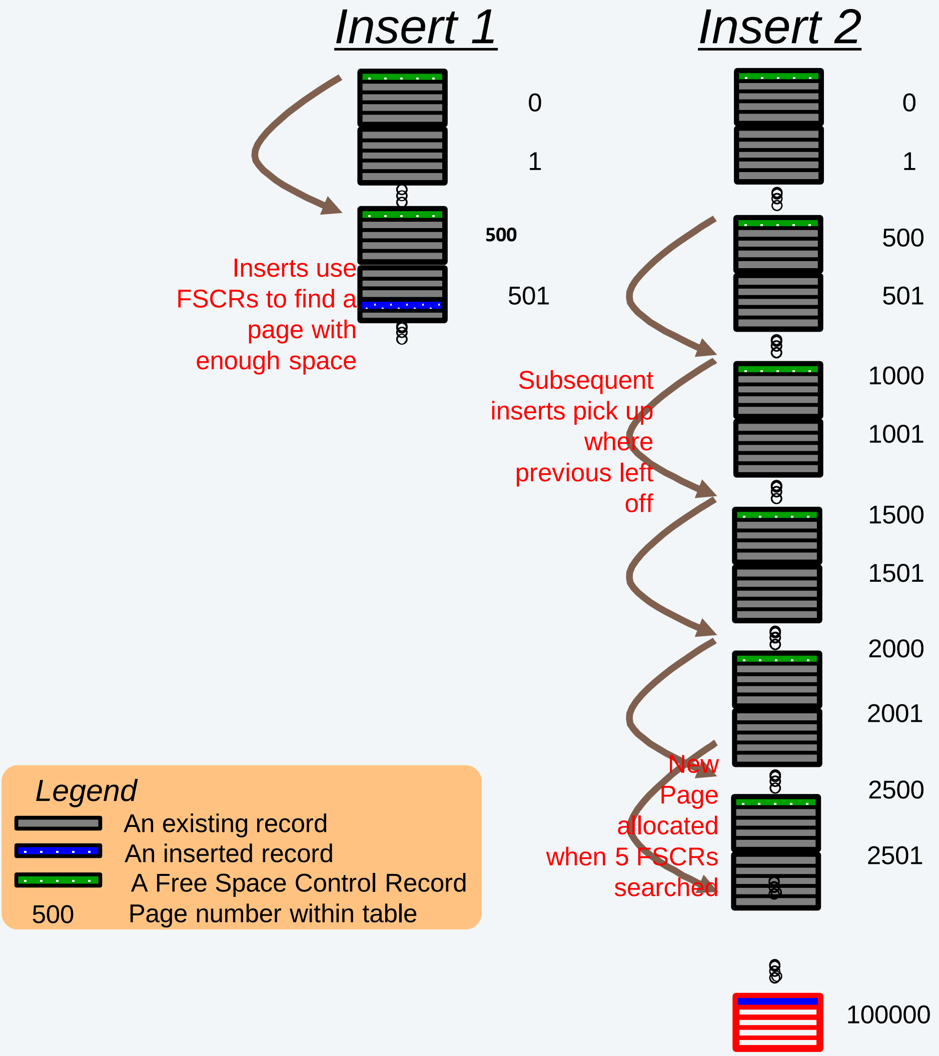
- Default INSERT search algorithm:
  - Use the Free Space Control Records (FSCRs) to find page with enough space
    - Even if an FSCR indicates that a page has enough free space, that space may not be usable if it is "reserved" by an uncommitted DELETE from another transaction
    - Ensure transactions COMMIT frequently; otherwise uncommitted freed space will not be reusable
  - Search 5 FSCRs (by default)
    - if there is no page with enough space, append record to end of table

INSERT Processing (& Space Mgt) 2/2

- DB2MAXFSCRSEARCH=<num> registry variable limits the number of FSCRs visited for an INSERT
  - Start with the default (5) for DB2MAXFSCRSEARCH, as it is designed for most workloads
  - Increase it to favor more aggressive space reuse, or, for extremely large tables
  - Decrease it to favor INSERT speed

- Each search starts at the FSCR where last search ended

- Once the entire table has been searched: we append without searching, until space is created elsewhere in table, via DELETE, for example


Insert 1
0
1
500
501
Inserts use FSCRs to find a page with enough space
Insert 2
0
1
500
501
1000
1001
1500
1501
2000
2001
2500
2501
100000
Subsequent inserts pick up where previous left off
New Page allocated when 5 FSCRs searched
Legend
An existing record
An inserted record
A Free Space Control Record
500 Page number within table

Inserts 3 through n

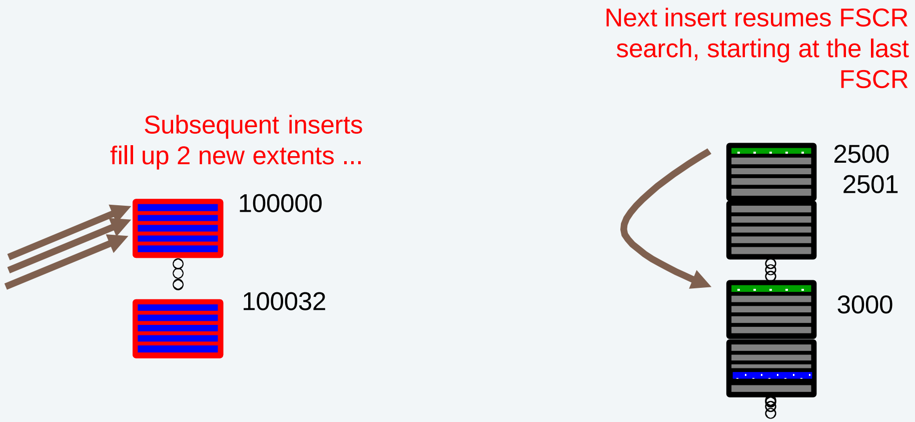
Insert n+1

Tips

Db2MAXFSCRSEARCH=<num> registry variable limits the # of FSCRs visited for an INSERT

Default of 5 works well for typical workloads

- Increase it to favor more aggressive space reuse, or, for extremely large tables
- Decrease it to favor INSERT speed
- Special value of -1 means unlimited FSCR search

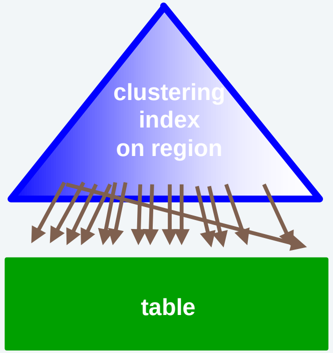

Subsequent inserts fill up 2 new extents ...
100000
100032
Next insert resumes FSCR search, starting at the last FSCR
2500
2501
3000

Space Management & Clustering

Other search algorithm options:

- Use ALTER TABLE APPEND ON (avoids searching and maintenance of FSCRs)
  - Tip: use APPEND ON for tables that only grow (eg journals)

- Use a clustering index on the table (CREATE INDEX ON T1 .... CLUSTER)

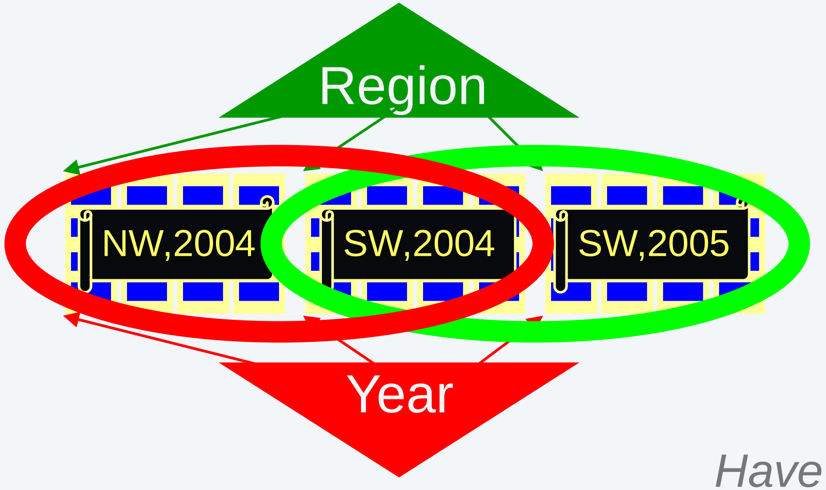


  - Db2 tries to insert records on the same page as other records with similar index key values, resulting in more efficient range scans and prefetching
  - If there is no space on that page, it tries the surrounding 500 pages, then reverts to the default search algorithm but uses a worst-fit, instead of first-fit approach (to establish a new 'mini' clustering area)
  - Tips:
    - Use a clustering index to optimize queries that retrieve multiple records in index order, as it results in less physical I/Os
    - When a clustering index is defined, use ALTER TABLE PCTFREE nn before load or reorg. This leaves nn% free space on the table's data pages after load and reorg, and increases the likelihood that the clustering insert algorithm will find free space on the desired page

Multidimensional Clustering

Divides the table up into ‘extents’ and ensures that each record in an extent contains the same value in all interesting dimensions

- Extent = consecutive group of pages, big enough for efficient I/O (typically 32 pages; 4 in the eg below)
- Queries in all dimensions benefit
- This clustering is always maintained by Db2; it never degrades



SELECT * FROM Sales WHERE Region = SW

- 2 big block I/Os to retrieve pages containing region SW
- All sequential I/O

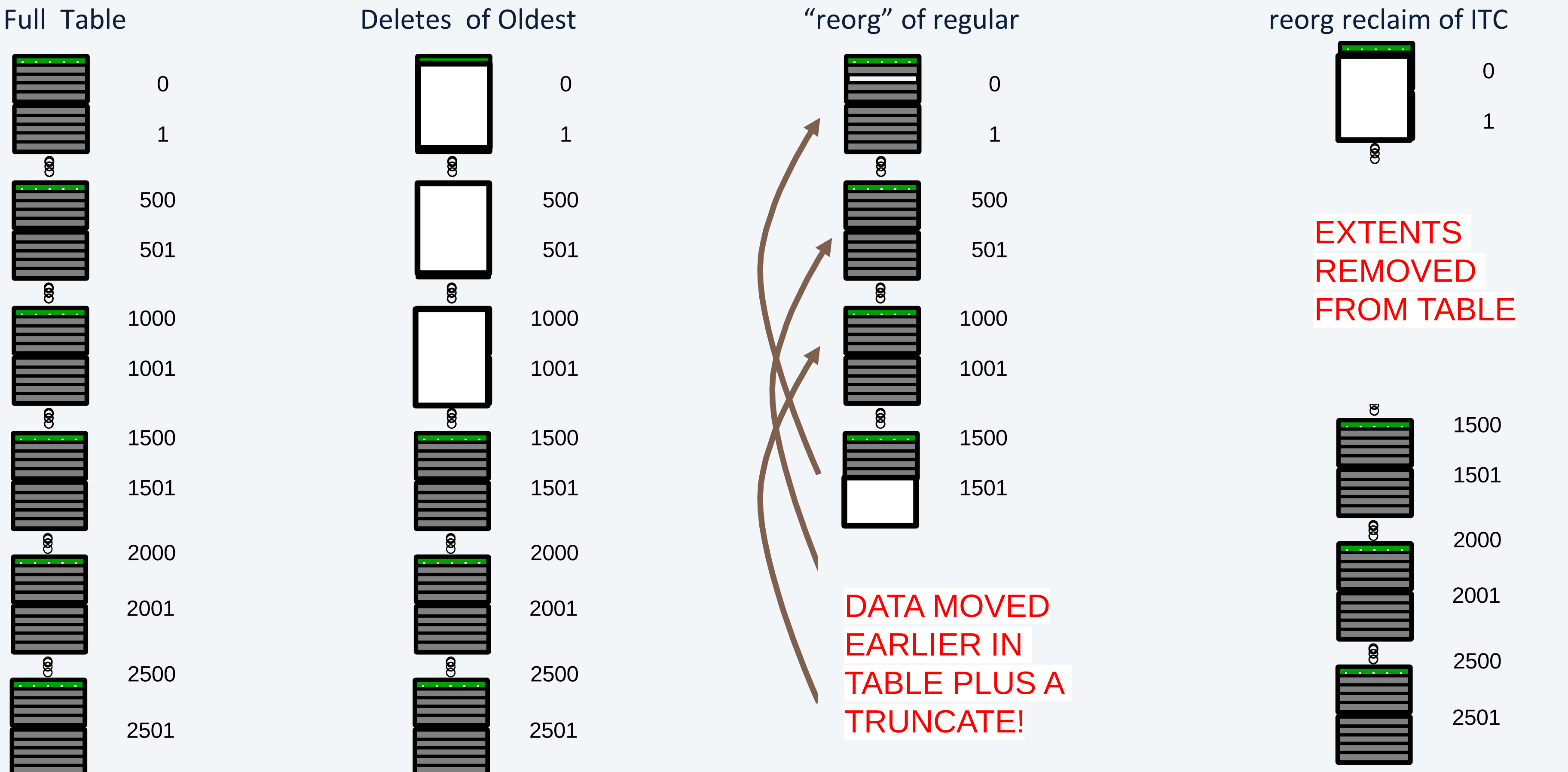
SELECT * FROM Sales WHERE Year = 2004

- 2 big block I/Os to retrieve pages containing year 2004
- All sequential I/O

Have your cake and eat it too !

ITC “reorg” benefits

Simply an append mode table where you can “reclaim” from front to create a sliding window.



EXTENTS REMOVED FROM TABLE

DATA MOVED EARLIER IN TABLE PLUS A TRUNCATE!

Space Mgt & Clustering : Hints / Tips / Reminders

Make effective use DB2MAXFSCRSEARCH

- Large values (or -1) to favor space reuse and reorg avoidance
- Small values to favor INSERT speed
-

If range scans are predominant use clustering to optimize their performance

- MDC or Clustering index
- Use the design advisor to assist with definition
-

APPEND mode can be useful in isolated scenarios to optimize INSERT speed

- However if/when mass deletes occur you either need to instead use ITC tables or have an alternative strategy

Range Partitioned Tables

Short and long forms

Partitioning column(s)

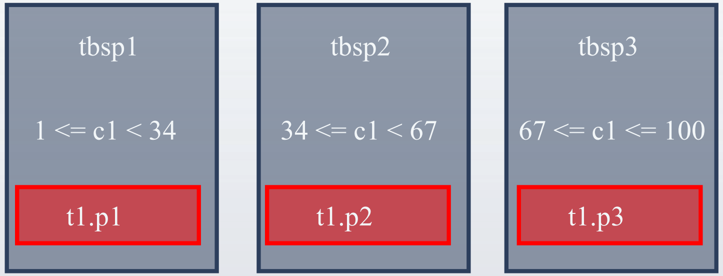
- Must be base types (e.g. No LOBS, LONG VARCHARS)
- Can specify multiple columns
- Can specify generated columns

Notes

- SQL0327N The row cannot be inserted because it is outside the bounds
- Special keywords, MINVALUE, MAXVALUE can be used to specify open ended ranges, e.g.:

  ```
  CREATE TABLE t1 …
  (STARTING(MINVALUE)
  ENDING(MAXVALUE) …
  ```

V11.1 added per partition REORG



Short Form

```
CREATE TABLE t1(c1 INT) IN tbsp1, tbsp2, tbsp3
  PARTITION BY RANGE(c1)
  (STARTING FROM (1) ENDING100) EVERY (33))
```

Long Form

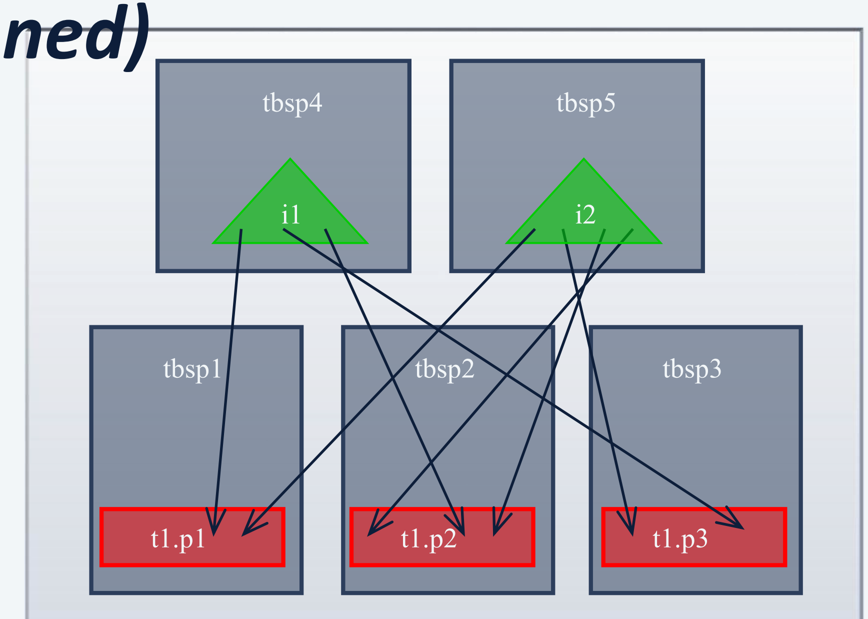
```
CREATE TABLE t1(c1 INT)
  PARTITION BY RANGE(a)
  (STARTING FROM (1) ENDING(34)   IN tbsp1,
  ENDING(67)                      IN tbsp2,
  ENDING(100)                     IN tbsp3)
```

Indexes : *Global (non-partitioned)*

Indexes can be *global* RIDs in index pages contain 2-byte partition ID

Each index is in a separate storage object

- By default, in the same tablespace as the first data partition
- Can be created in different tablespaces, via
  - INDEX IN clause on CREATE TABLE (default is tablespace of first partition)
    - Note: INDEX IN clause works for MDC indexes ('block' indexes)
  - New IN clause on CREATE INDEX
- Recommendation
  - Place indexes in LARGE tablespaces
  - ** Per partition reorg with global indexes is not yet supported.

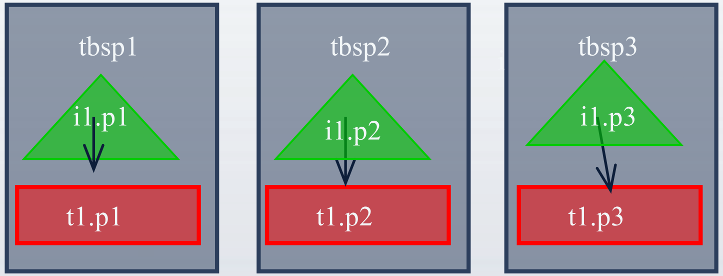


```
CREATE TABLE t1(c1 INT, c2 INT, …)
IN tbsp1, tbsp2, tbsp3
INDEX IN tbsp4
PARTITION BY RANGE(c1)
(STARTING FROM (1) ENDING100) EVERY (33))

CREATE unique INDEX i1 (c2)
CREATE INDEX i2 (c3) IN tbsp5
```

Indexes : *Local (partitioned)*

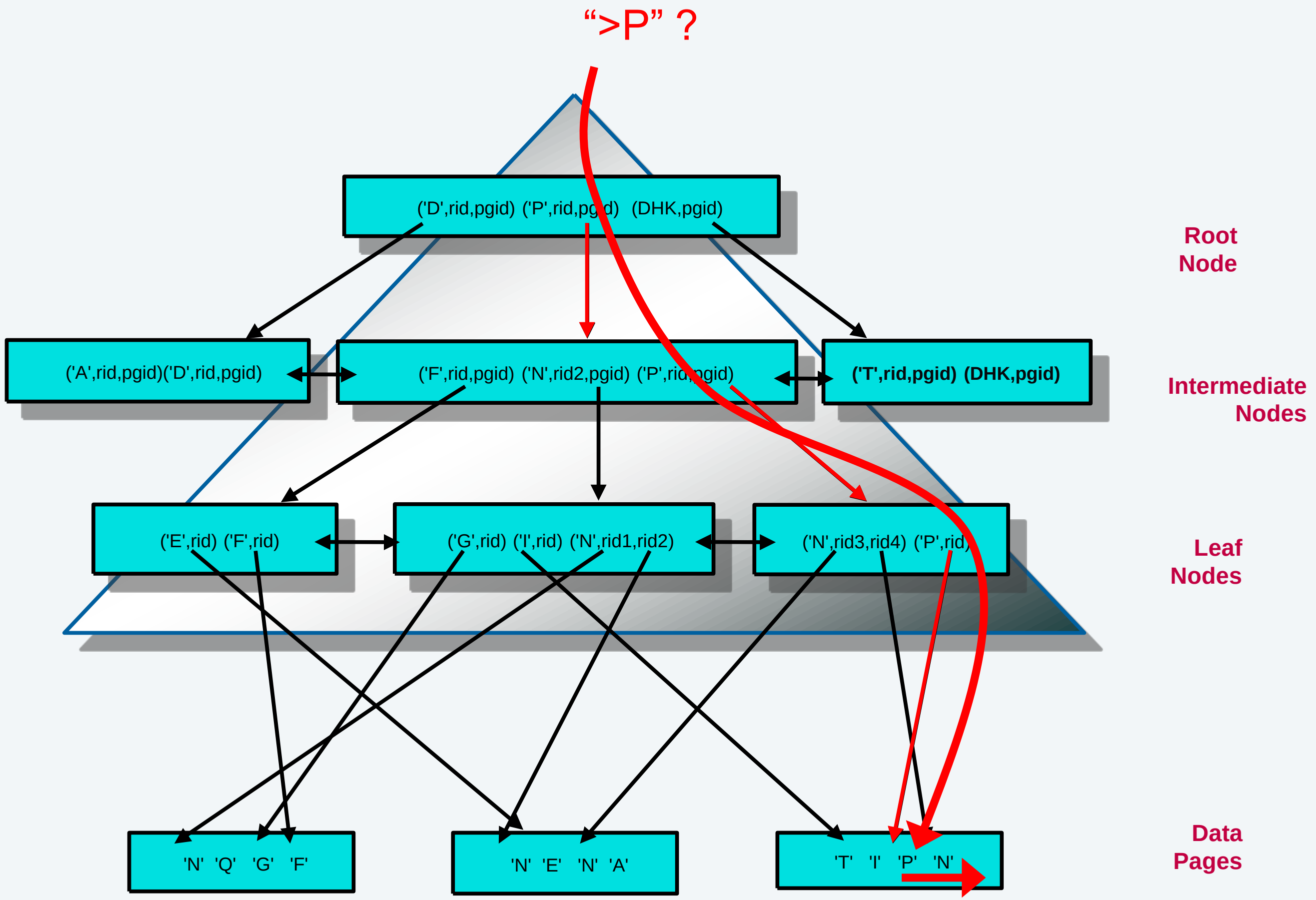
Indexes can also be *local* RIDs

All local indexes in single storage object (like a non-partitioned table)



```
CREATE TABLE t1(c1 INT, c2 INT, …)
IN tbsp1, tbsp2, tbsp3
PARTITION BY RANGE(c1)
(STARTING FROM (1) ENDING100) EVERY (33))

CREATE INDEX i1(c1) PARTITIONED
```

B+ Indexes

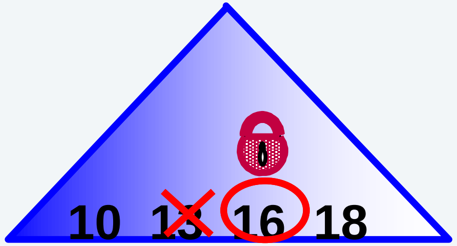

“>P” ?
('D',rid,pgid) ('P',rid,pgid) (DHK,pgid)
Root Node
('A',rid,pgid)('D',rid,pgid)
('F',rid,pgid) ('N',rid2,pgid) ('P',rid,pgid)
('T',rid,pgid) (DHK,pgid)
Intermediate Nodes
('E',rid) ('F',rid)
('G',rid) ('I',rid) ('N',rid1,rid2)
('N',rid3,rid4) ('P',rid)
Leaf Nodes
'N' 'Q' 'G' 'F'
'N' 'E' 'N' 'A'
'T' 'I' 'P' 'N'
Data Pages

Type 2 Indexes and "Pseudo-Deleted Keys"

APP 1: DELETE FROM T1
WHERE C1=13

APP 2: INSERT INTO T1
VALUES (13,....)
APP 2: SELECT ... FROM T1
WHERE C1>10

Type 1 Indexes and Next Key Locking

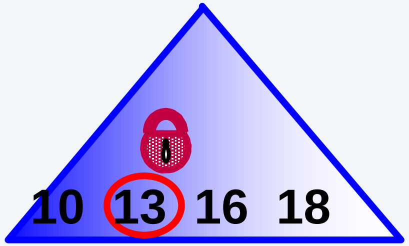


APP 1: DELETE FROM T1
WHERE C1=13

APP 2: INSERT INTO T1
VALUES (13,....) → *prevents unique violations*
APP 2: SELECT ... FROM T1
WHERE C1>10 → *prevents dirty reads*

APP 3: INSERT INTO T1
VALUES (12,....) → *allows false conflict*

Type 2 Indexes with Pseudo-Deletes



→ *prevents unique violations*

→ *prevents dirty reads*

APP 3: INSERT INTO T1
VALUES (12,....) → *no false conflict !!*

When are Pseudo-Deleted Keys Freed?

During INSERTS

- If such a cleanup might avoid the need to split the page

During subsequent deletes

- If a new delete results in all keys on the page being marked as deleted, an attempt will be made to find another page that only contains pseudo-deleted keys and for which all the deletes are committed; if such a page is found, it will be removed from the index tree

Any rebuild of the index including those resulting from:

- REORG TABLE (not using the INPLACE option)
- REORG INDEXES ALL
- IMPORT with REPLACE
- LOAD with the INDEXING MODE REBUILD option

When the REORG INDEXES command with the CLEANUP option is specified

- CLEANUP ONLY PAGES :
- Searches for and frees indexes pages on which all keys are marked deleted and known to be committed
- CLEANUP ONLY ALL :
- Frees not only index pages on which all keys are marked deleted and known to be committed, but also removes keys marked deleted and known to be committed on pages that contain some undeleted keys

Tip : REORG INDEXES … CLEANUP is more efficient faster than a full index REORG

- Done in-place (no separate object built, and not object-switch phase)