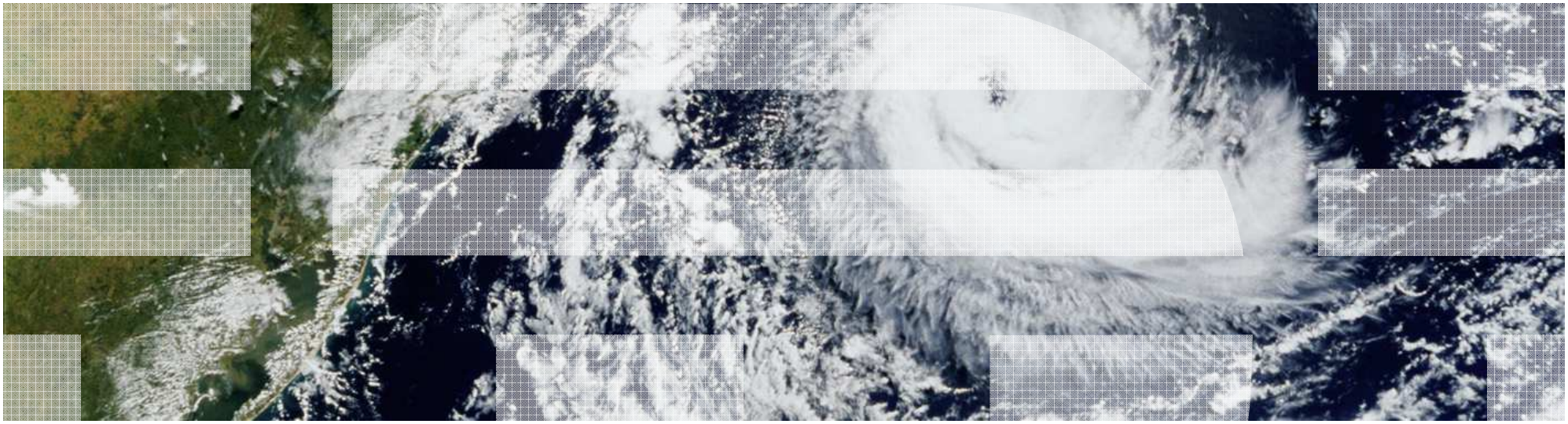
DB2 10 for z/OS – What's In It For Me?



Disclaimers and Trademarks

Agenda

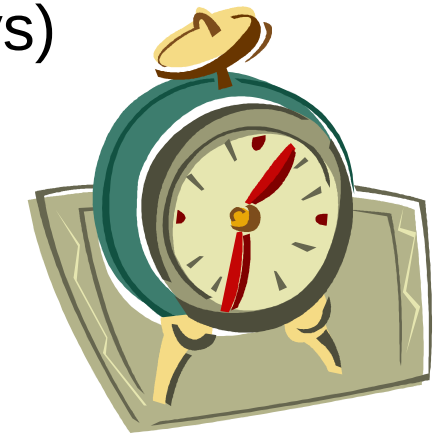
- Online Schema
- Index Enhancements
- Buffer Pools
- Universal Table Space
- Auto Compress
- Hash Table Access
- Other Enhancements
- Temporal Data – A Quick Overview

Without Online Schema

- Unload data
- Drop base tables
- Create base tables
- Create indexes
- Create views
- Define authorizations
- Load data (Copy, Stats, Check)
- Rebind

Data available for read

Data unavailable (minutes, hours, days)



Data available

Online Schema – Before DB2 10

- Table or Column Changes:
  - Increase column size within datatype integer, float
  - Change to expand datatype: char, numeric
  - Change varchar to / from char
  - Rename Table/Column
- Index Changes
  - Add new column to index
  - Change index column type
  - Rename Index
- Dynamic Partitions
  - Add Partition
  - Rotate Partition

DB2 10 Online Schema Enhancements

- Table space changes – deferred
  - Page Size – UTS only
  - DSSIZE – UTS only
  - SEGSIZE – UTS only
  - MAXPARTITIONS
  - Convert single table simple/segmented into PBG
  - Convert classic partitioned table into PBR
  - Convert PBR to PBG
  - Convert Classic Partitioned/PBR/PBG to Member Cluster
  - Table Space is put into Advisory REORG pending
- Index changes – deferred
  - Page size
  - Index is put into Advisory REORG pending

One way only!

DB2 10 Online Schema Enhancements (cont.)

- Deferred Alters are materialized at the next table/index space Reorg
  - Before REORG, pending changes can be dropped via ALTER TABLESPACE/INDEX with the new DROP PENDING CHANGES clause
  - Quiesce applications using the DBD, plan/package locks during the SWITCH phase
  - Plans/packages/dynamic statement cache invalidated when table space type altered
- Alter Buffer Pool with the same page size is effective immediately
  - No longer need to STOP the table or index space in data sharing
  - Use DRAIN(ALL) to quiesce applications
  - The new bufferpool will be used after commit

The ALTER Statement and Pending DDL

- The statement is validated

DB2 semantically validates the statement and checks the authorisation

- Assuming all checks out ok:
  - The statement is put on the to-do-list
  - The table space is placed in Advisory REORG Pending (non-restrictive) - AREOR
  - The statement completes with SQL Code +610 to indicate the AREOR state

The to-do list is stored in new catalog table, SYSPENDINGDDL

SYSIBM.SYSPENDINGDDL

DBNAME	TSNAME	DBID	PSID	...	OPTION_ KEYWORD	OPTION_ VALUE	...	STATEMENT_ TEXT

REORG and the Materialization of Deferred Alters

- Pending DDL is materialized during the SWITCH phase
- SYSPENDINGDDL entries are removed
- Stats are collected
  - The default is TABLE ALL INDEX ALL UPDATE ALL HISTORY ALL unless overridden
  - Warning message DSNU1166I is issued to indicate that some partition statistics may no longer be accurate (COLGROUP, KEYCARD, HISTOGRAM …)
- SYSCOPY records are created to record the materialization of the deferred ALTER
- The Advisory REORG Pending state is reset.

Deferred ALTER restrictions

- Mixing immediate and deferred options in an ALTER statement is not allowed (SQL Code -20385 is issued)
- Many immediate DDL statements are not allowed while there is pending DDL awaiting materialization (SQL Code -20385 is also issued in this case)
  - These include CREATE/DROP/ALTER TABLE for a table with pending DDL

```
DSNT408I SQLCODE = -20385, ERROR:  THE STATEMENT CANNOT BE PROCESSED BECAUSE
         THERE ARE PENDING DEFINITION CHANGES FOR OBJECT DB1.TS1 OF TYPE
         TABLESPACE (REASON 2)
```

- Image Copies before the materializing REORG are no longer valid for recovery
  - REPORT RECOVERY enhanced to use '#' to indicate status of image copy

```
TIMESTAMP = 2010-08-03-20.07.20.813333,  IC TYPE = #F#,  SHR LVL = R,  DSNUM    = 0000,  START LRSN =0000090539AE
 DEV TYPE  =         ,                    IC BACK = FC,   STYPE   = T,  FILE SEQ = 0000,  PIT LRSN  = 0000090539E2
 LOW DSNUM = 0001,   HIGH DSNUM = 0001,   OLDEST VERSION = 0000,   LOGICAL PART = 0000,   LOGGED = Y,   TTYPE = C
 JOBNAME   = DB2R8L  ,     AUTHID =   DB2R8   ,     COPYPAGESF = -1.0E+00
 NPAGESF   = -1.0E+00           ,                  CPAGESF = -1.0E+00
 DSNAME    = DB0BI.DB1.TS1.N00001.CXZO6WVC                , MEMBER NAME =         , INSTANCE = 01, RELCREATED = M
```

Online Schema – 'Immediate' DDL

- Alter Index to add included columns
  - Can add non-key columns (INCLUDE columns)
  - The index is marked as Rebuild Pending unless the new columns are added to the new table within the same COMMIT scope, in which case the index is placed in Advisory REORG Pending
  - Packages are not invalidated, but BIND or REBIND is strongly recommended
- Alter PBR/PBG to Hash Access
  - The table is not accessible via hash access as the overflow index is marked as Rebuild Pending
  - New rows cannot be inserted unless the overflow index is rebuilt or until REORG
- ALTER LOB to use in-line LOBs
  - REORG required to convert existing LOBs to in-line

DB2 10 Index Improvements

- Parallel index update at insert
  - For tables with more than two indexes
    - Applies for Member Cluster or APPEND tables with two or more indexes
  - Uses I/O parallelism

- INLCUDE columns for indexes:

```
ALTER INDEX ix1 ADD INCLUDE (c4);
ALTER INDEX ix1 ADD INCLUDE (c5);
```

  - Add non-key columns
  - Unique index only
  - The index is marked as Rebuild Pending unless the new columns are added to the new table within the same COMMIT scope, in which case the index is placed in Advisory REORG Pending
  - Packages are not invalidated, but BIND or REBIND is strongly recommended
  - Potential to reduce the number of indexes
  - Packages dependent on indexes dropped still invalidated

DB2 10 Index Improvements (cont.)

- Improve index look-aside performance
  - Cache the last accessed leaf page
- Improve index split performance
  - Asymmetric index split, depending on insert pattern
- Cache the index root page in the buffer pool during OPEN
  - Eliminate one GETP/RELP on each index access
- List prefetch for disorganized indexes – forward scan only
  - Reduces the need for REORG
- Improve performance for Referential Integrity and Nested Loop Join
  - Enable index look-aside and sequential detection
- CPU reduction for index access especially with NOT PADDED indexes (VARCHAR)
- New IFCID 359 to monitor index split activity

Buffer Pool Enhancements

- Use 1MB real storage frame, instead of 4K, for PGFIX=YES buffer pools
  - Reduces cpu overhead when accessing pages in BP (fewer TLB misses)
  - Requires z10, z196 or zEnterprise
- Prefetch and deferred writes are offloaded to zIIP
- Avoid BP scans in data sharing
  - Switch from no shared interest to shared and vice versa
  - Data set close and –STOP DB2
  - Significantly improves performance for large BPs
- In-memory tables/Indexes
  - New option PGSTEAL=NONE on ALTER BUFFERPOOL
  - Prefetch loads data into the buffer pool at data set open
  - Ensure BP size is large enough to cache all data/index pages
  - BP is managed in FIFO (First In First Out) order, not LRU
    - Disables all prefetch requests
  - Query optimizer uses zero I/O cost for tables and indexes in this buffer pool
  - Recommendation: use AUTOSIZE=NO
  - Recommendation: use CLOSE=NO for tables and indexes

Auto-compress (Compress on Insert)

- Insert can now build compression dictionary on the fly
  - Applies to COMPRESS YES tables without a dictionary
  - Compresses rows without needing a REORG or LOAD
  - Also applies to online LOAD inserts and to the MERGE SQL statement
- The dictionary is built by an asynchronous DB2 task
  - The driver for this is based on DATASIZE in RTS catalog table
  - Uses UR to perform a relational scan of the existing data rows
- Ensure you use the default SYSTEMPAGES YES option for image copies
- The dictionary needs to have been built before the UNLOAD utility can read any data rows

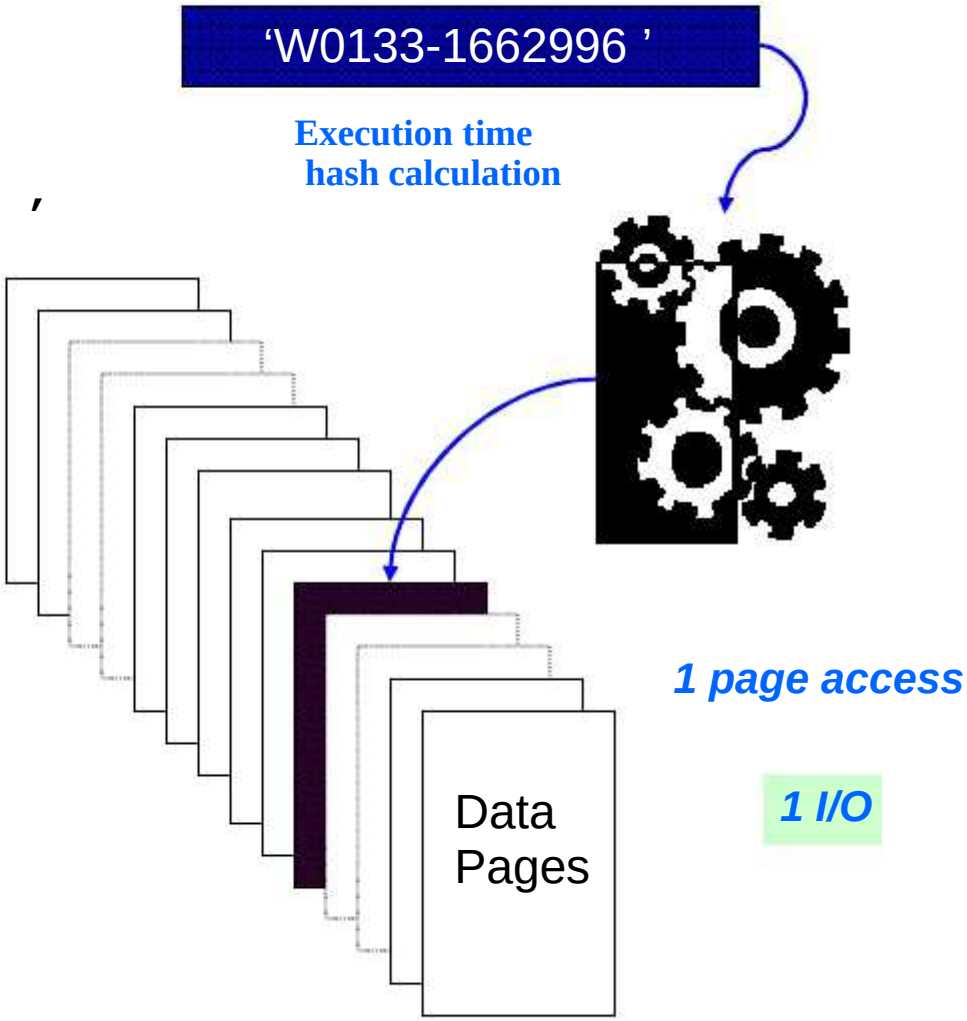
Hashed Tables

```
SELECT * …WHERE
   ITEMNO = 'W0133-1662996 '
```

Locate the data row
by hashing the key value.

-The hash Key must be unique
-Reduced page visits
-Reduced CPU & elapsed time
-Possibly eliminate an index

-Tradeoff: extra space used



Hashed Table Access Considerations

- Tables
  - Hash access is only useful for tables:
    - With a unique key
    - Queried by applications (such as OLTP) needing single row access via the unique key or IN list
    - With known approximate size
- Applications
  - Hash access path will be chosen when:
    - The SELECT statement or IN list uses equal predicates on all hash key columns
    - No range predicates
    - DB2 Optimzer determines whether hash access is suitable
- Syntax
  - CREATE TABLE … ORGANIZE BY HASH UNIQUE (itemno) HASH SPACE 2G
  - ALTER PBR/PBG into Hash
    - REORT required for Hash Access

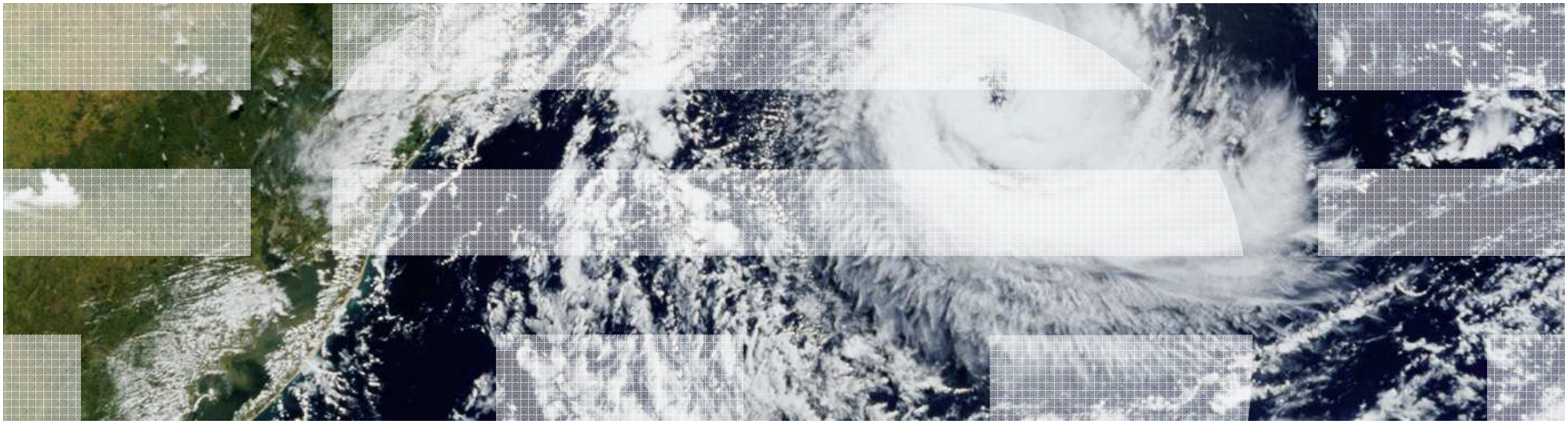
Universal Table Space Enhancements

- MEMBER CLUSTER now supported for UTS
  - ALTER support: deferred materialized with REORG
- ALTER can add new partitions to PBG
  - Allows multiple partitions to be built when the PBG is created
  - Can now use DSN1COPY to copy data to a PBG target
- PBR is the default for new ranged partitioned tables
  - Need to specify SEGSIZE 0 to create classic ranged partitioned table space
  - Can make classic range partitioned table space the default with new ZPARM DPSEGZ=0
    - Needs NUMPARTS clause on DDL

COPY/RECOVER Enhancements

- Ability to create a consistent COPY with SHRLEVEL CHANGE
  - Does not quiesce applications via drain
  - Implemented by data set level FlashCopy
- COPY, LOAD, and REORG extended to provide ability to take backups using data set level FlashCopy
- Data set level FlashCopy backups can be used as input to:
  - RECOVER, COPY, DSN1COPY, DSN1PRNT
- RECOVER extended to support point-in-time recovery via rollback using the DB2 log

Temporal Data – A Quick Overview



Problem of managing different versions of application data

- Application programmers and database administrators have always struggled with managing different time-varying versions of application data
- New regulatory laws require maintaining historical versions of data for years
- Every update and delete of data requires old data be copied into history tables
- Existing approaches to application level versioning complicates table design and adds complexity and error prone code to applications

Introduction to Temporal Data

- Temporal data allows you to see data in its current state, as it was in the past, and even at some future point in time.
- Temporal data:
  - Provides the ability to see data values at a given point in time.
  - Provides the ability to see data values over a past, current or future period of time.
  - Supports history and/or auditing queries.
  - Supports the concepts of Business Time and System Time.
- Benefits of temporal data:
  - Can move temporal data maintenance from the application layer to the database layer.
    - Consistent handling of temporal data.
  - Reduction in application development time.
  - Can run current applications with no code change.

Temporal Table Overview

- Business Time (Effective Dates, Valid Time, In/Out-dates)
  - Every row has two timestamp or date columns set by the application.
    - Begin time: when the business deems the row valid.
    - End Time: when the business deems row validity ends.
  - Query at current, any prior, or future point/period in business time.
- System Time (Assertion Dates, Knowledge Dates, Transaction Time, Audit Time).
  - Every row has another pair of timestamps set by DB2.
    - Begin time: when the row was inserted in the DB2.
    - End Time: when the row was modified/deleted.
    - Modified rows begin time is the modification time.
  - Query at current or any prior point/period in system time.
- Bi-temporal (Two dimensional history, Two dimensional tombstoning/milestoning).
  - Inclusion of both System Time and Business Time in row.

Benefits of DB2 10 Temporal Data Implementation

- Can manage application data based upon time at the table level.
- Can specify search criteria based upon the time the data existed or was valid.
  - Simplifies DB2 application development where data versioning is required.
- Customers can satisfy new compliance laws faster and cheaper because DB2 will automatically manage the different versions of data.
- Performance expectations:
  - SELECT of current data should perform similar to tables without data versioning
  - INSERT and UPDATE of current data would perform slower than table not performing data versioning
  - SELECT of old data may be slower

System Time with Versioning

- Application on current data preserved
  - Insert, Update, Delete, Select unchanged
  - Triggers unchanged
  - Constraints unchanged
- DB2 preserves current application by moving history rows to separate table
- SELECT w/o AS OF behaves as if AS OF SYSTEM_TIME DATE CURRENT DATE transparently added
- DB2 translates AS OF times into predicates on system time column pair
- INSERT/UPDATE/DELETE behave as if FOR PORTION OF SYSTEM_TIME FROM TRANSACTION TIME TO END OF TIME transparently added
- DB2 optimizer presents current and history tables as one, pulls answers from both as needed and presents as one answer set

SYSTEM_TIME Period Example (DDL)

```
CREATE TABLE     policy_info
(policy_id CHAR(4)              NOT NULL,
coverage INT                    NOT NULL,

sys_start  TIMESTAMP(12)    NOT NULL        GENERATED ALWAYS AS ROW BEGIN,
sys_end    TIMESTAMP(12)    NOT NULL        GENERATED ALWAYS AS ROW END,
create_id TIMESTAMP(12)     GENERATED ALWAYS AS TRANSACTION START ID,
PERIOD   SYSTEM_TIME(sys_start, sys_end));

CREATE TABLE     hist_policy_info
(policy_id CHAR(4)              NOT NULL,
coverage INT                    NOT NULL,
sys_start  TIMESTAMP            NOT NULL,
sys_end    TIMESTAMP            NOT NULL,
create_id TIMESTAMP(12));

ALTER TABLE                     policy_info
ADD VERSIONING HISTORY TABLE    hist_policy_info;
```

BUSINESS_TIME Period DDL

- To create a temporal table with BUSINESS_TIME period, the table needs to be defined with:
  - A start column defined as TIMESTAMP(6) NOT NULL or DATE NOT NULL
  - An end column defined as TIMESTAMP(6) NOT NULL or DATE NOT NULL
  - a period BUSINESS_TIME specified on the two above columns.
    - example of period specification: PERIOD BUSINESS_TIME(col1, col2)
- The temporal table with BUSINESS_TIME period can also be defined to have a unique attribute that is unique for a period of time. For example:
  - UNIQUE (col1, BUSINESS_TIME WITHOUT OVERLAPS)
  - PRIMARY KEY (col1,col2, BUSINESS_TIME WITHOUT OVERLAPS)
- The BUSINESS_TIME WITHOUT OVERLAPS can be on the CREATE TABLE, ALTER TABLE, or CREATE UNIQUE INDEX statements.
  - BUSINESS_TIME WITHOUT OVERLAPS must be the last expression specified
  - It will add, in ascending order, the end column and start column of the period BUSINESS_TIME to the index key and have special support to enforce that there are no overlaps in time given the rest of the index key expressions.

BUSINESS_TIME Period DML

- To query from a temporal table with BUSINESS_TIME period, the table-reference of the FROM clause is extended to specify that BUSINESS_TIME period data is requested.
- The FROM clause extension includes:
  - table-name FOR BUSINESS_TIME AS OF date-expression *or* table-name FOR BUSINESS_TIME AS OF timestamp-expression
  - table-name FOR BUSINESS_TIME FROM date-expression1 TO date-expression2 *or* table-name FOR BUSINESS_TIME FROM timestamp-expression1 TO timestamp-expression2
    - note that the second expression is not inclusive
  - table-name FOR BUSINESS_TIME BETWEEN date-expression1 AND date-expression2 *or* tablename FOR BUSINESS_TIME BETWEEN timestamp-expression1 AND timestamp-expression2
    - note that the second expression is inclusive

BUSINESS_TIME Period DML (cont.)

- Temporal tables with BUSINESS_TIME period, the UPDATE and DELETE statement have been enhanced to allow updating and deleting based upon a period of time. New clause is specified as follows:
  - FOR PORTION OF BUSINESS_TIME FROM date-expression1 TO date-expression2
  - FOR PORTION OF BUSINESS_TIME FROM timestamp-expression1 TO timestamp-expression2
- The new update or delete clause will update/delete rows as usual if the row's period is totally contained in the period specified in the FROM and TO clauses.
- Whenever a row is not totally contained by these values, only the part of the row that is contained between the FROM and TO clauses are updated or deleted. This may cause additional inserts of rows into the table as the original row needs to be split into multiple rows to contain the old values for the period that is not specified

BUSINESS_TIME Period Example (DDL)

```
CREATE TABLE  policy_info
(policy_id        CHAR(4)          NOT NULL,
coverage          INT              NOT NULL,
bus_start         DATE             NOT NULL,
bus_end           DATE             NOT NULL,
PERIOD            BUSINESS_TIME(bus_start, bus_end);

CREATE UNIQUE INDEX  ix_policy
ON policy_info (policy_id, BUSINESS_TIME WITHOUT OVERLAPS);
```

DB2 10 Temporal Table Benefits

- SYSTEM_TIME period
  - Provides database managed START and END date maintenance providing non-overlapping time periods with no holes
  - Provides automatic management of data movement from temporal to historical table
  - Provides automatic rewrite of queries on temporal table to include a UNION to history table
  - Minimizes performance impact of temporal support when no rows qualify from the history table for a query
- BUSINESS_TIME period
  - Provides a new unique index clause to prevent overlapping time periods for a business key
  - Provides database management of the temporal data when an UPDATE or a DELETE statement is executed with a FOR PORTION OF BUSINESS_TIME clause that does not exactly align with an existing row
- BI_TEMPORAL table
  - Provides support for application defined business periods with non-destructive data modification.
  - This combines all of the features provided for SYSTEM_TIME period and BUSINESS_TIME period

Temporal Table Considerations

- The business decision to support temporal characteristics in a database table should be made with a clear understanding of the considerable costs implicit in this choice
  - Increased storage size for the table. The increase is proportional to the UPDATE and DELETE rate for this table.
  - Degraded database performance for UPDATE, DELETE, and query
- Queries will nearly always include two range predicates for start and end columns for each type of period defined on the table (four range predicates for bi_temporal). At most, one of those predicates will be index matching.
- UPDATES and DELETES that are not precisely aligned with the start and end columns of a single row will generate multiple operations. Often this will be four operations.
  - YSTEM_TIME period will require inserting one or multiple rows in the history table
- Application complexity access and modifying a temporal table will be increased
- If your business requirements make it essential to support temporal characteristics, the DB2 database support provided in DB2 10 will provide considerable relief from the complexities of supporting these requirements at an application level.