PostgreSQL Internals Uncovered: Enhancing Logical Replication and Introducing Columnar Indexing

Vigneshwaran C

Software Lead Developer
Fujitsu

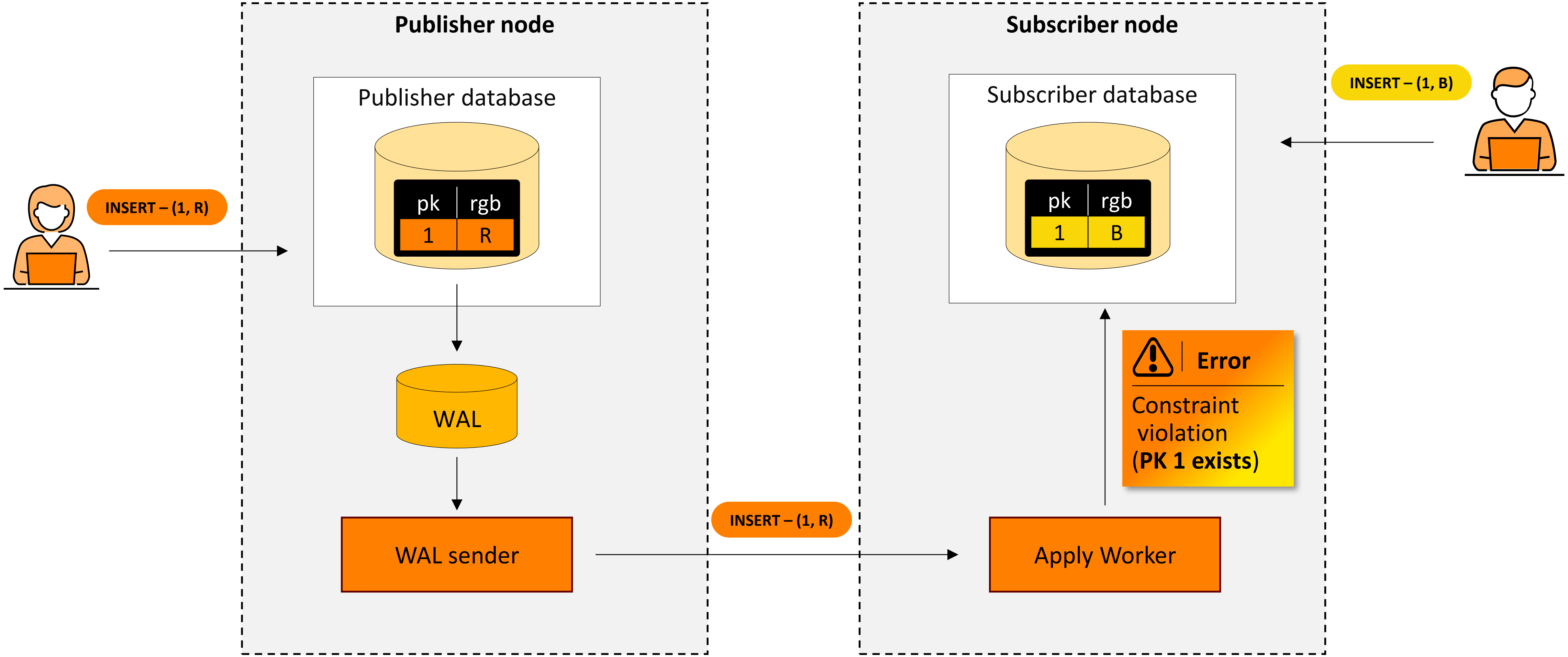
PG Down Under 2025

Agenda

- Conflicts
- Conflict detection and logging
- Conflict resolution
- Conflict history
- Upgrading replica using pg_createsubscriber
- Internals of pg_createsubscriber
- Sequence Replication
- Columnar Store
- Vertical Clustered Index

Conflicts – logical replication

- A conflict occurs when incoming changes from the publisher disagree with the current state of data on the subscriber.
- It can happen because of local writes on subscriber or subscriber has multiple publishers.



Conflicts – conflict handling prior to PG17

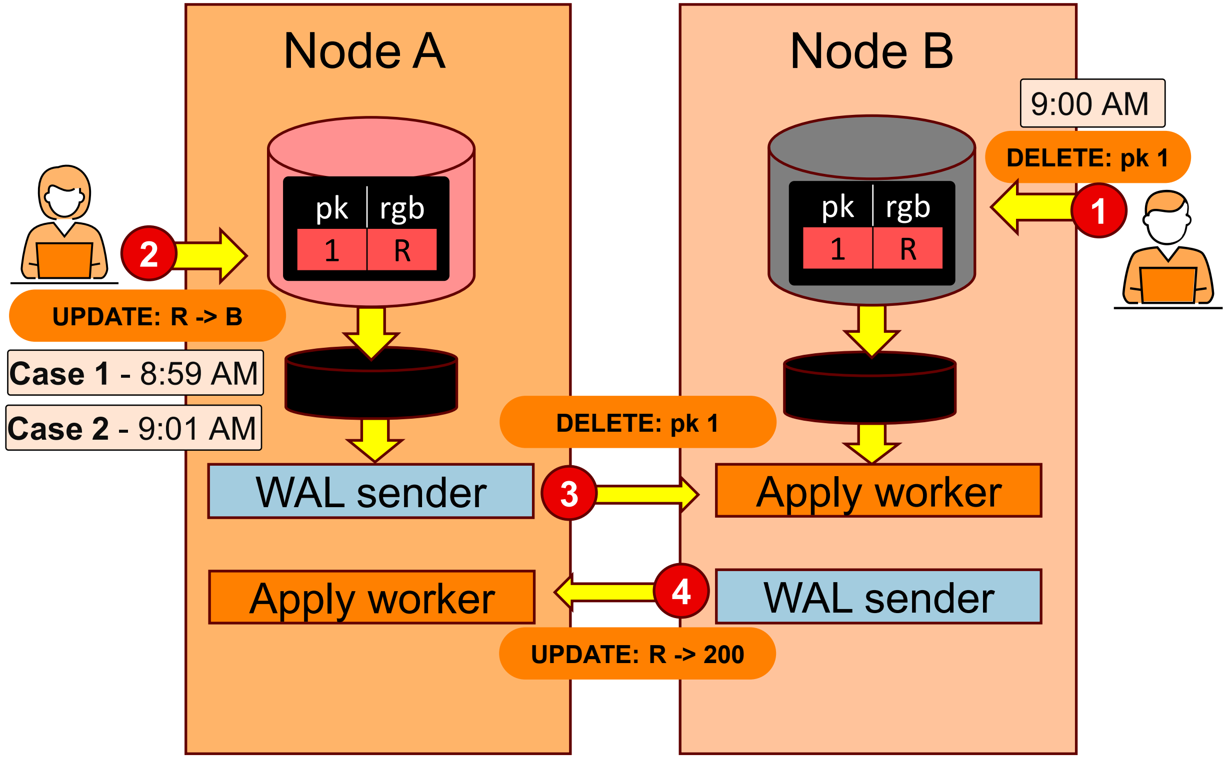
- On insert conflict, apply worker **gives error, exits and is restarted**
- Apply worker becomes **stuck in error loop**
- Conflict needs to be handled **manually**
- `pg_replication_origin_advance()` could be used to advance past conflicting transaction but it is **difficult to figure out the LSN of conflicting transaction**.
- Update and delete conflicts are **not reported and are silently skipped**
- **Minimal information** in error log message.

Conflict detection and logging in PG18

INSERT_EXISTS **UPDATE_EXISTS**	Inserting/Updating a row that violates a NOT DEFERRABLE unique constraint.
UPDATE_ORIGIN_DIFFERS **DELETE_ORIGIN_DIFFERS**	Updating/deleting a row that was previously modified by another origin. Note that this conflict can only be detected when track_commit_timestamp is enabled on the subscriber
UPDATE_MISSING **DELETE_MISSING**	The tuple to be updated/deleted was not found
MULTIPLE_UNIQUE_CONFLICTS	Inserting or updating a row violates multiple NOT DEFERRABLE unique constraints

Necessity of UPDATE_DELETED

- Detecting `update_deleted` conflicts is important for achieving eventual consistency in a bidirectional cluster.



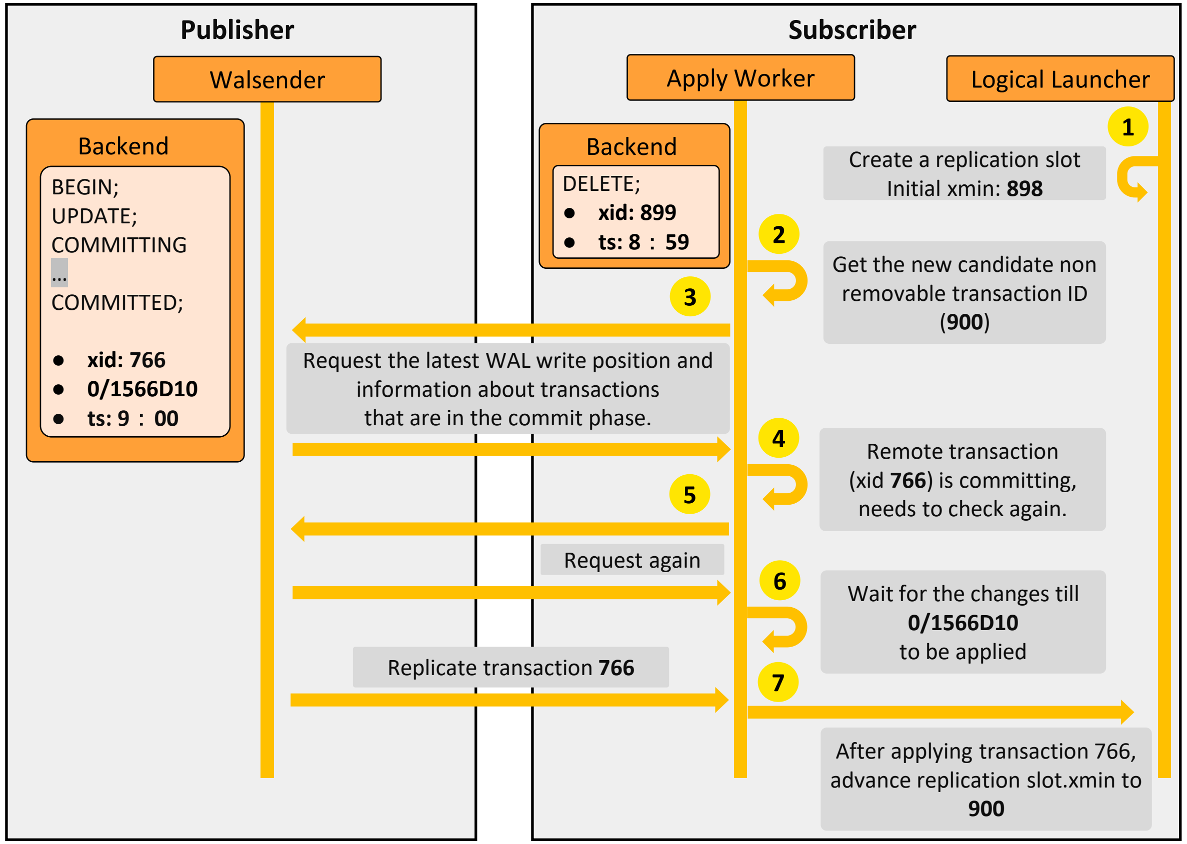
- Assume that the user wants to maintain consistency by keeping the changes with the latest commit timestamp. (last-update-win strategy)
- When applying the remote update in step 3, it is found that the target row does not exist.
- Following the last-update-win strategy:
  - In **case 1**, the remote update occurs earlier than the local deletion, so the update on node B should be skipped.
  - In **case 2**, the remote update occurs later, so a new row

pk	rgb
1	B

should be inserted into node B.

  - **Without detecting `update_deleted`, the case 2 could not be solved properly.**

Conflicts – Retain dead tuples



- A replication slot is created to retain dead tuples.
- Dead tuples are retained until all remote transactions that occurred concurrently with the tuple DELETE are applied and flushed locally.
- The apply worker requests the walsender to get the latest WAL write position and information about transactions that are committing.
- Do not proceed if there are concurrent remote transactions that are committing, because these transactions might have been assigned an earlier commit timestamp.
- Wait till the transactions are committed in the publisher and the transactions are applied on the subscriber.
- If the update data is not found in the subscriber, perform a scan again with SnapshotAny to identify the deleted record and report update_deleted conflict.

Conflicts - Retention duration

- If a subscriber with retain_dead_tuples = true lags significantly, the retained dead tuples could consume excessive disk space.
- A new subscription parameter, **max_retention_duration**, is introduced. If the apply worker's lag (measured by how far back its non-removable transaction ID is) exceeds this threshold, the apply worker **stops retaining conflict information**.
- The conflict detection slot's xmin is set to InvalidTransactionId, effectively disabling the retention mechanism to allow VACUUM to proceed with cleanup and prevent disk exhaustion. A new catalog column (subretentionactive) persists this disabled state across restarts.

```
postgres=# CREATE SUBSCRIPTION <subname> CONNECTION <conninfo> PUBLICATION <pubname>  WITH
                   (max_retention_duration = 1000);

postgres=# ALTER SUBSCRIPTION <subname> SET (max_retention_duration = 1);
```

Conflicts - Resolution Framework

- Automatic handling of conflicts based on configuration & type.
- Built-in resolvers reduce user effort
- It can be used to achieve eventual consistency in bidirectional replication.
- Flexible control: users specify resolvers per conflict type in subscription DDL.

```
postgres=# CREATE SUBSCRIPTION <subname> CONNECTION <conninfo> PUBLICATION <pubname>
CONFLICT RESOLVER
            (conflict_type1 = resolver1, ...);

postgres=# ALTER SUBSCRIPTION <subname> CONFLICT RESOLVER
            (conflict_type1 = resolver1, ...);

postgres=# ALTER SUBSCRIPTION <subname> RESET CONFLICT RESOLVER ALL;
```

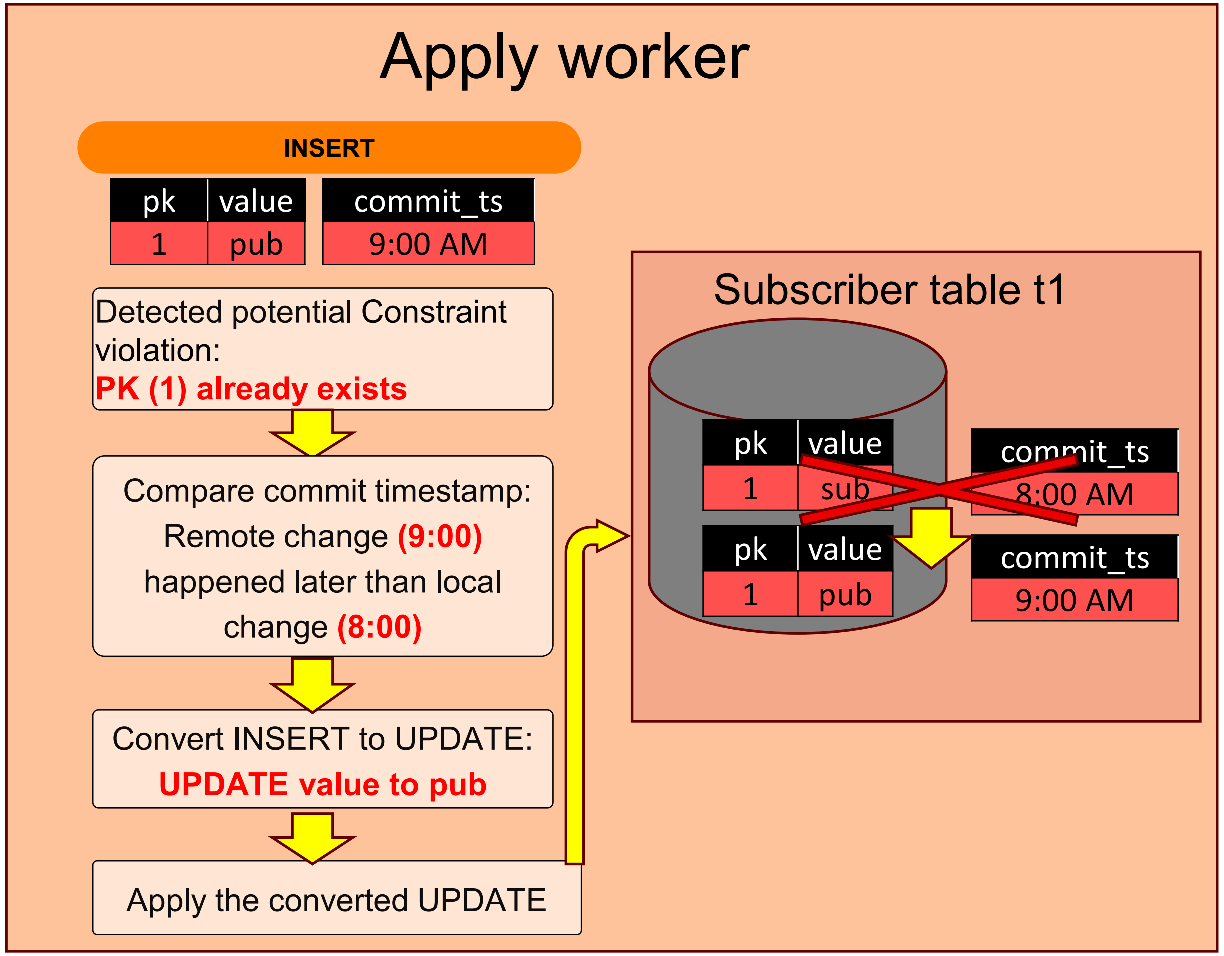
- The default resolvers for all conflict types are aligned with existing PostgreSQL behavior for consistency.

Conflicts - Built-in conflict resolvers

Resolver	Description	Applicable conflicts
apply_remote	The remote change is applied	● insert_exists ● update_exists ● multiple_unique_conflicts ● update_origin_differs ● delete_origin_differs
apply_or_skip	Remote change is converted to INSERT and is applied. If the complete row cannot be constructed from the info provided by the publisher, then the change is skipped.	● update_missing ● update_deleted
apply_or_error	Like above but raise an ERROR if cannot convert the remote change to INSERT.	● update_missing ● update_deleted
keep_local	The local version of row is retained.	● insert_exists ● update_exists ● multiple_unique_conflicts ● update_origin_differs ● delete_origin_differs
last_update_wins	The change with later commit timestamp wins.	● insert_exists ● update_exists ● update_origin_differs ● delete_origin_differs ● update_deleted
error	Replication is stopped; manual action is needed.	Can be used for any conflict type.

Basic flow for **last_update_wins**

- Resolve **insert_exists** conflict using **last_update_wins**



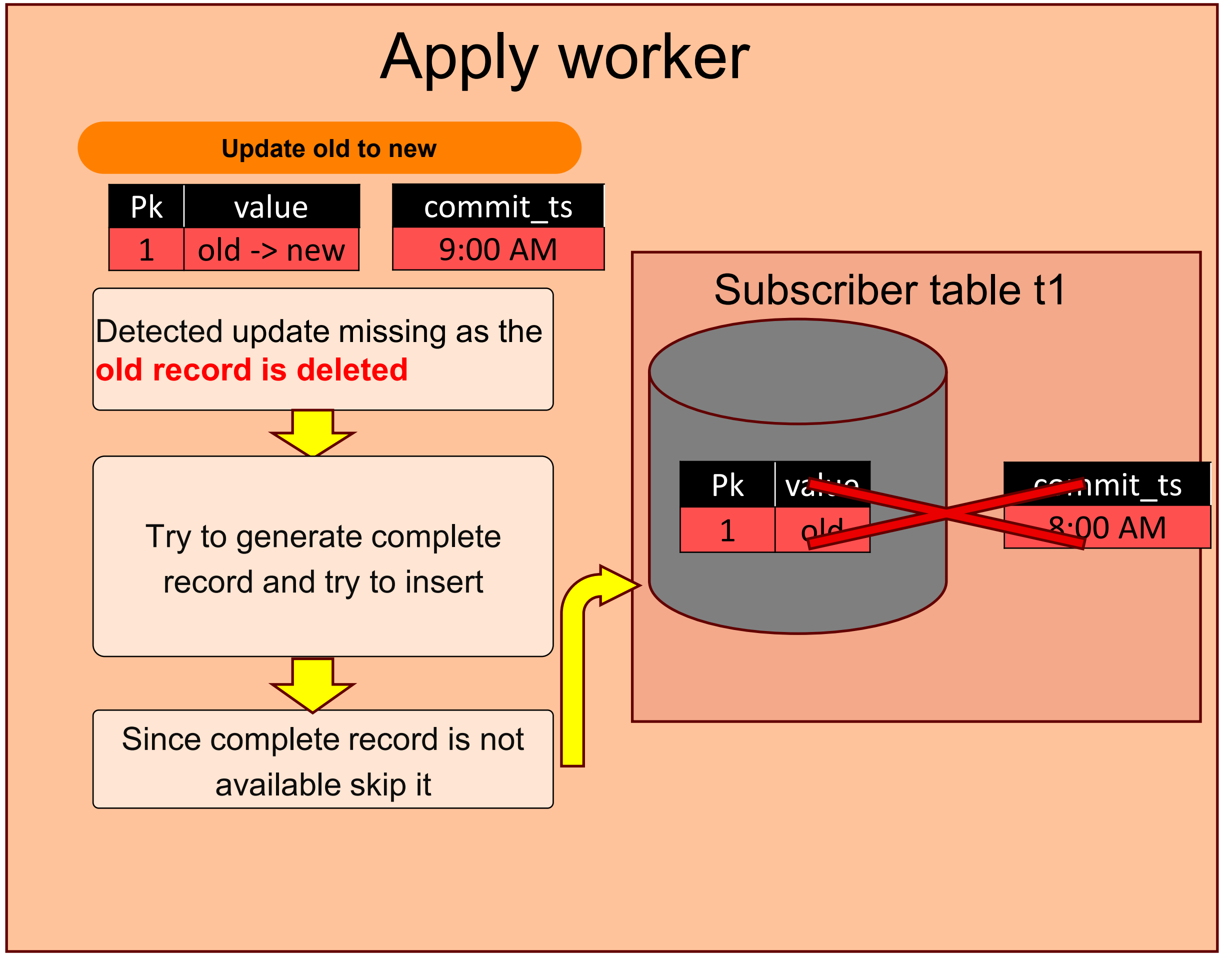
- Before applying the remote INSERT, the apply worker checks if any conflicting tuples with the same primary key already exist.
- If conflicting tuples exist, the apply worker gets the commit timestamp from the commit_ts module using the xmin of the conflicting tuple.

```
track_commit_timestamp = ON
```

- If the conflicting tuple's commit timestamp is later than that of the remote transaction, the remote INSERT is skipped.
- Otherwise, it creates an UPDATE to change the value column of the conflicting tuples to match the remote value.

Basic flow for **apply_or_skip**

- Resolve update_delete conflict using **apply_or_skip**



- Consider a case where the old record is deleted in the subscriber
- Subscriber then receives the update. Subscriber identifies that the matching record is not found in the subscriber.
- Publisher does not send unchanged toasted column value
- It identifies the conflict as update_missing as the record has been deleted
- Under apply_or_skip, the apply worker tries to regenerate the complete record but it cannot generate as the complete record was not sent, it did not send the toast column values as they were not changed.
- It will skip this and continue.

Conflicts - Viewing conflict history

1/2

FUJITSU

- Currently conflict only visible in logs; hard to analyze
- With this feature a new user specified table will be created to store all the details of a conflict:
  - Table and the tuple details
  - conflict type
  - remote XID, LSN and commit_time_stamp.
  - local XID, LSN and commit_time_stamp
  - origin info
- A new subscription option is added to create the user specified table(which must not already exist) as –

```
postgres=# CREATE SUBSCRIPTION sub CONNECTION 'dbname=postgres port=5432' PUBLICATION pub
WITH(conflict_log_table=myschema.my_conflict_table);
```

Conflicts - Viewing conflict history

2/2

- Conflict reported in LOG

```
LOG:  conflict detected on relation "public.test": conflict=update_origin_differs

DETAIL:  Updating the row that was modified locally in transaction 762 at 2025-09-04 16:48:29.099432+05:30.
        Existing local row (1, 10); remote row (1, 20); replica identity (a)=(1).

CONTEXT:  processing remote data for replication origin "pg_16391" during message type "UPDATE" for replication target
relation "public.test" in transaction 757, finished at 0/01711C10
```

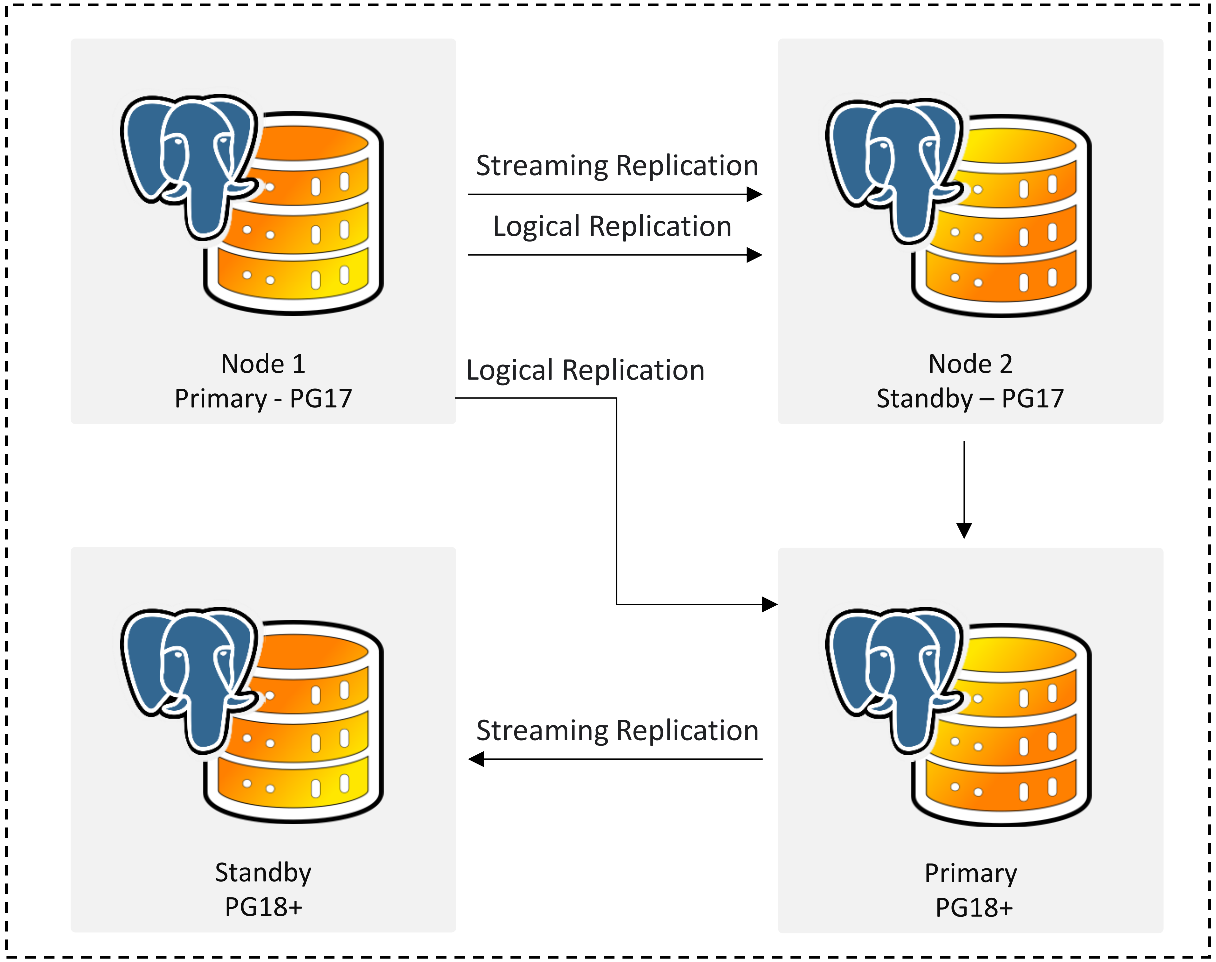
- Conflict details stored in the table:

```
postgres-# select * from myschema.my_conflict_table;
relid                | 16385
local_xid            | 762
remote_xid           | 757
local_lsn            | 0/00000000
remote_commit_lsn    | 0/01711C10
local_commit_ts      | 2025-09-04 16:48:29.099432+05:30
remote_commit_ts     | 2025-09-04 16:48:34.122196+05:30
table_schema         | public
table_name           | test
conflict_type        | update_origin_differs
local_origin         |
remote_origin        | pg_16391
key_tuple            | {"a":1,"b":20}
local_tuple          | {"a":1,"b":10}
remote_tuple         | {"a":1,"b":20}
```

Upgrade - Online upgrade of replica

FUJITSU

Here we want to upgrade streaming replication clusters from PG17 to PG18+…



1 Run pg_createsubscriber

```
$ pg_createsubscriber -D node2 …
```

2 Stop and upgrade

```
$ pg_ctl -D node2 stop
$ pg_upgrade -d node2 …
```

3 Stop Node1

```
$ pg_ctl -D node1 stop
```

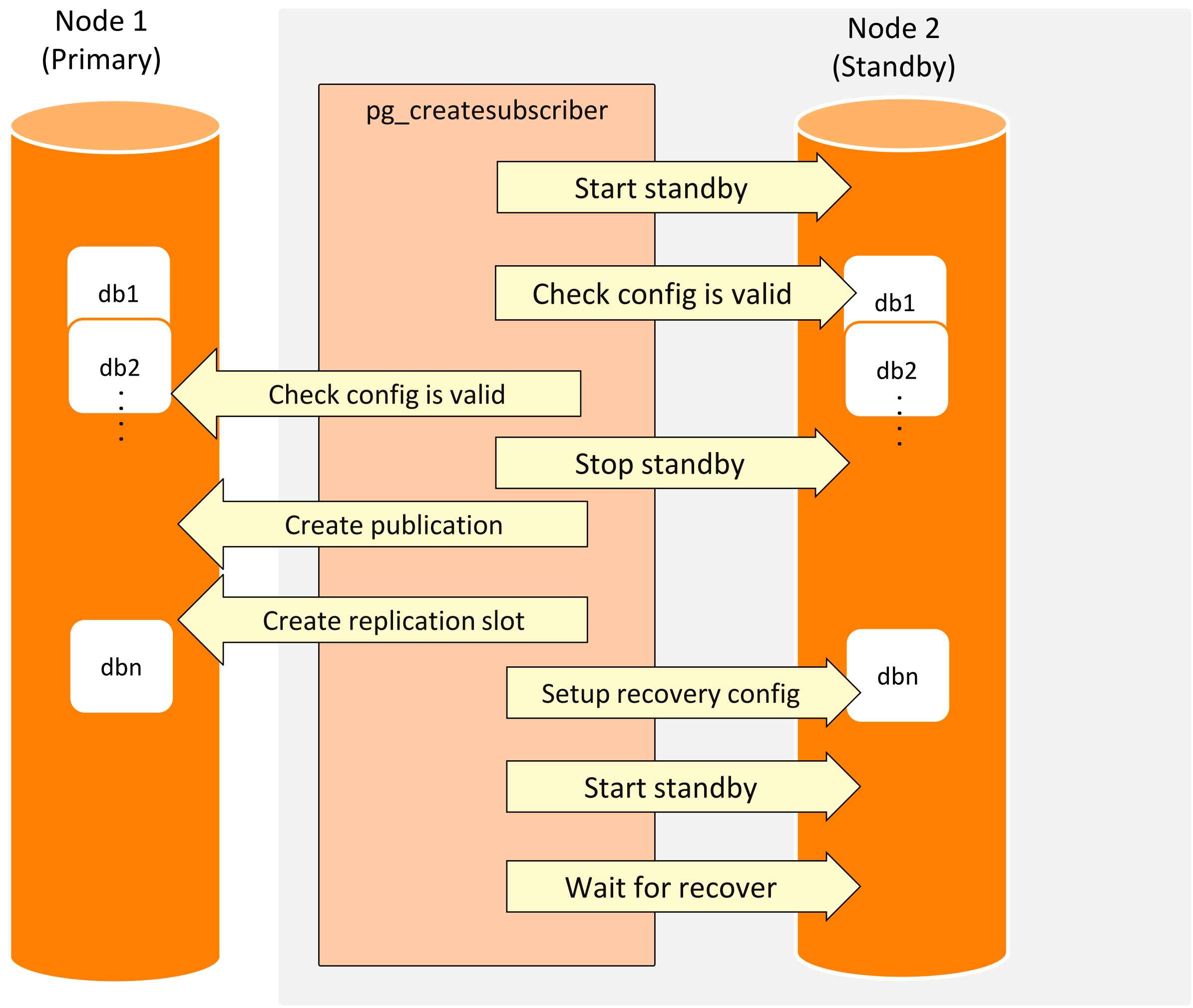
4 Run pg_basebackup

```
$ pg_basebackup -D node2_upgraded -R…
```

5 Drop subscriptions

```
DROP SUBSCRIPTION subXX…
```

Internals of pg_createsubscriber



- Start standby
- Check standby configs are ok
- Check publisher configs are ok
- Stop standby server
- Create publication
- Create replication slot
- Write the lsn from above in recovery_target_lsn
- Start the standby server
- Wait until recovery reaches LSN
- Create subscription with copy_data as false and slot_name from above created slot
- Set the origin to the lsn specified in recovery_target_lsn

New/Upcoming features to pg_createsubscriber

FUJITSU

- Support all option
  - Users had to run pg_createsubscriber separately for each database, specifying -d
  - User runs one command with --all, and subscriptions are created for all matching databases.
- Support enable-two-phase option
  - The --enable-two-phase option enables support for replicating prepared transactions. Subscription will be created with two_phase option.
- Support clean option
  - The --clean option in pg_createsubscriber enhances logical replication setup by removing any pre-existing replication artifacts on the subscriber—specifically, existing publications that might interfere with the new configuration.

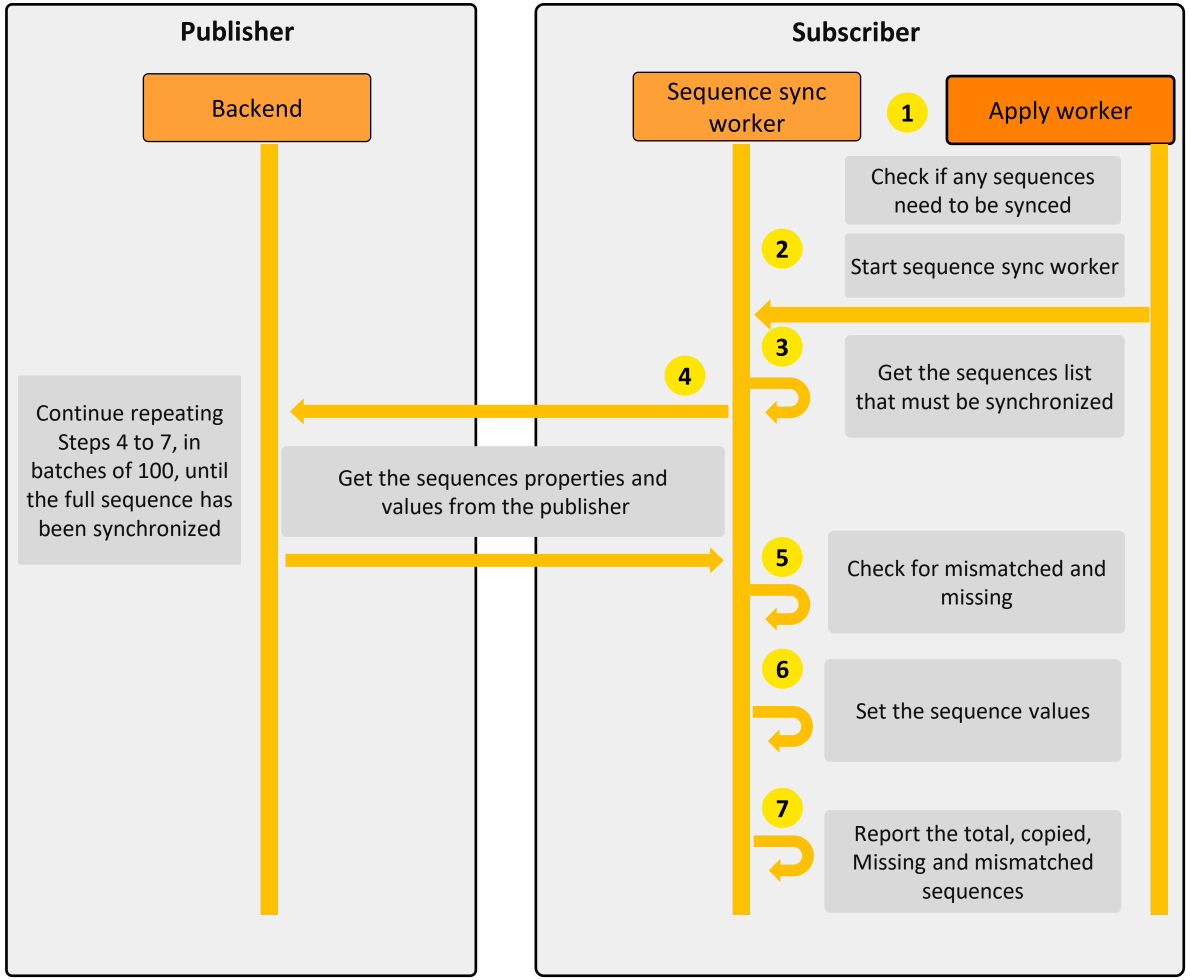
New/Upcoming features to pg_createsubscriber

- Validation of max_slot_wal_keep_size
  - Misconfigured max_slot_wal_keep_size (e.g., 1GB) could silently result in missing WAL and replication failure. pg_createsubscriber emits a warning if max_slot_wal_keep_size is not -1, reducing risk up front.
- Support existing-publication
  - The --existing-publication option, allowing users to leverage pre-configured publications instead of the default FOR ALL TABLES approach. This enhancement provides greater flexibility and fine-grained control over replication configuration.

Sequence replication

- Currently logical replication is used for upgrade of physical replication cluster, but one of the problem is that the table data will be replicated but the sequences are still at the initial values, requiring some custom solution that moves the sequences forward enough to prevent duplicities after upgrade.
- Sequences can now be synced from publisher to subscriber.
- New command to support:
  - CREATE PUBLICATION for ALL SEQUENCES;
    - A subscription created for publication for “ALL SEQUENCES” allow subscriber to fetch and sync the sequences from publisher
  - ALTER SUBSCRIPTION <sub> REFRESH REPLICATION SEQUENCES;
    - REFRESH re-synchronize all the existing sequence value with the updated sequence value from the publisher

Sequence replication - Synchronizing sequence



1. Check if any sequences should be synced
2. Start sequence sync worker
3. Sequence sync worker will get the sequence list that must be synced
4. Get the above sequence properties and sequence values from publisher
5. Check for mismatched and missing sequences
6. Update the subscriber sequence with publisher values
7. Report the total, copied, missing and mismatched count
8. Repeat step 4 to step 7 in batches of 100

Sequence replication - Identifying out of sync sequences

- The page LSN of the sequence from the publisher is also captured and stored in pg_subscription_rel.
- This LSN will reflect the state of the sequence at the time of synchronization.
- By comparing the current LSN of the sequence on the publisher (via pg_sequence_state()) with the stored LSN on the subscriber, users can detect if the sequence has advanced and is now out-of-sync.
- This comparison will help determine whether re-synchronization is needed for a given sequence.

Use case: Primarily designed to simplify **upgrades**.

Columnar Store (1/3)

- **High-performance data retrieval**
  - I/O can be reduced by scanning only columns needed for data analysis
  - All columns must be read in a row-oriented database

```
SELECT Product, Date FROM purchase_history;
```

Purchase_ID	Product	Price	Date
111	Ice cream	120	4/1
111	Cake	450	4/1
333	Chocolate	170	4/15
444	Ice cream	210	4/27

Need to read all columns

Purchase_ID	Product	Price	Date
111	Ice cream	120	4/1
111	Cake	450	4/1
333	Chocolate	170	4/15
444	Ice cream	210	4/27

Scan only needed columns

Columnar Store (2/3)

FUJITSU

- **Efficient compression**
  - Long-term database can be effectively compressed

Purchase_ID	Product	Price	Date
111	Ice cream	120	4/1
111	Cake	120	4/1
333	Ice cream	120	4/1
444	Ice Cream	210	4/27

Purchase_ID	Product	Price	Date
111 *x* 2	Ice cream	120 *x* 3	4/1 *x* 3
333	Cake	210	4/27
444	Ice cream *x* 2		

Compression ratio is high because same patterns are repeated

Columnar Store 3/3

FUJITSU

- **Not optimized for INSERT/UPDATE/DELETE**
  - All columns must be read and updated

```
INSERT INTO purchase_history VALUES (555, 'Apple', 210, '5/1');
```

Purchase_ID	Product	Price	Date
111	Ice cream	120	4/1
111	Cake	450	4/1
333	Chocolate	170	4/15
444	Ice cream	210	4/27
555	Apple	210	5/1

Done at once

Purchase_ID	Product	Price	Date
111	Ice cream	120	4/1
111	Cake	450	4/1
333	Chocolate	170	4/15
444	Ice cream	210	4/27
555	Apple	210	5/1

Needs I/O per attribtues

Vertical Clustered Index (VCI) Proposal

- Columnar store function using column type index
- VCI allows you to:
  - Ability to aggregate data generated by OLTP in real time
  - Faster data retrieval
- Current functional range of VCI
  - Columnar Store
    - Conversion to column is asynchronous to avoid degradation of update performance
    - Refer to all data including update data using column format data and update difference information

VCI - INDEX Creation

- CREATE INDEX creates **Read Optimized Storage** for the specified row

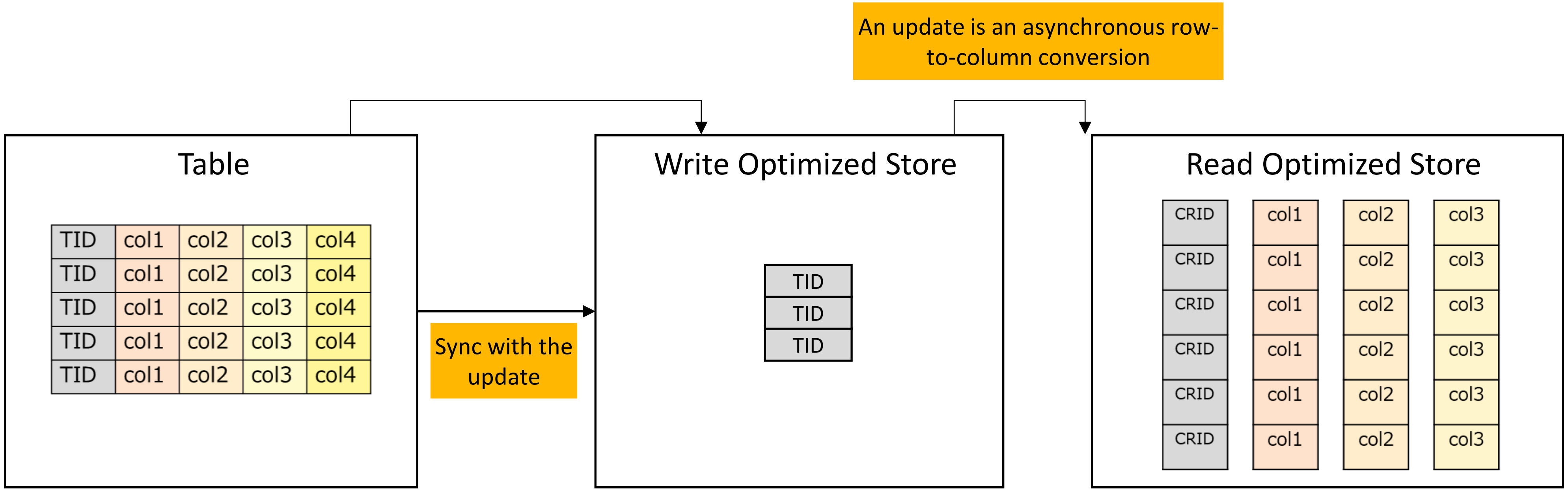
Purchase_ID	Product	Price	Date
111	Ice cream	120	4/1
111	Cake	450	4/1
333	Chocolate	170	4/15
444	Ice cream	210	4/27

```
CREATE INDEX vcindex ...
USING vci (Price, Date)…;
```

Read Optimized Store

Price	Date
120	4/1
450	4/1
170	4/15
210	4/27

How VCI Works



Write-Optimized-Store (WOS)

The row-wise data storage structure in VCI. It buffers data inserts, deletes and updates to VCI index so that writes to ROS contain a sufficient number of rows to amortize the cost of writing.

Read-Optimized-Store (ROS)

The column-wise data storage structure in VCI.

VCI - Searching Tables

- Find ROS data when executing SELECT query
- Ability to quickly search only product lines

Ex. tally up sales of goods

```
SELECT SUM(price) FROM purchase_history
WHERE date > '2025-03-31';
```

Read Optimized Store

Price	Date
120	4/1
450	4/1
170	4/15
210	4/27

VCI - Updating Tables

- When INSERT occurs, store TID in **Write Optimized Storage** and ROS is not updated.
- Only small-data inserts go to WOS, so updates are minimally impacted.

TID	Purchase_ID	Product	Price	Date
1	111	Ice cream	120	4/1
2	111	Cake	450	4/1
3	333	Chocolate	170	4/15
4	444	Ice cream	210	4/27
	555	Apple	198	5/1

`INSERT;`

Write Optimized Store

TID
5

Read Optimized Store

Price	Date
120	4/1
450	4/1
170	4/15
210	4/27

VCI - SELECT after INSERT

- Temporarily convert WOS data to **Local ROS**
- Search for Local ROS and ROS

```
SELECT SUM(price) FROM purchase_history WHERE date > '2025-03-31';
```

TID	Purchase_ID	Product	Price	Date
1	111	Ice cream	120	4/1
2	111	Cake	450	4/1
3	333	Chocolate	170	4/15
4	444	Ice cream	210	4/27
5	555	Apple	198	5/1

Write Optimized Store

TID
5

Local
Read Optimized Store

Price	Date
198	5/1

Read Optimized Store

Price	Date
120	4/1
450	4/1
170	4/15
210	4/27

VCI - ROS Update (Asynchronous)

- BG Worker propagates changes to ROS after a while

TID	Purchase_ID	Product	Price	Date
1	111	Ice cream	120	4/1
2	111	Cake	450	4/1
3	333	Chocolate	170	4/15
4	444	Ice cream	210	4/27
5	555	Apple	198	5/1

BG Worker

Write Optimized Store

TID
5

Read Optimized Store

Price	Date
120	4/1
450	4/1
170	4/15
210	4/27
198	5/1

Summary

FUJITSU

Distributes writes

- Conflict detection & logging - Concurrent writes to the same row on different nodes and logs them. This is crucial for maintaining data consistency across a distributed system.
- Built-in conflict resolvers -When a conflict is detected, automated rules can be applied to resolve it. Common strategies include "last committer wins" or "first committer wins," ensuring the system doesn't require manual intervention for every conflict.
- Conflict storage - Unresolved or logged conflicts are stored in a dedicated area, often a system table, allowing administrators to review and manually handle any conflicts that are not automatically resolved

High Availability of Logical Replica

- pg_createsubscriber - A PostgreSQL utility designed to automate the process of setting up a logical replica. It handles the initial synchronization of data from the publisher, including creating the subscription and replication slot.
- Synchronization of sequences - A key part of the synchronization process is ensuring that sequences on the replica have the same state and value as the publisher

Performance

- Vertical Clustered Index (VCI) - For efficient data analysis and aggregation

Thank you

Vigneshwaran C

Software Lead Developer
Fujitsu