DB 2

05 – Row Updates

Summer 2020

Torsten Grust
Universität Tübingen, Germany

1 ¦ Q_4 — Row Update

SQL probe Q_4 uses SQL DML statements (INSERT, UPDATE, DELETE) to alter the state of a table:

```
INSERT INTO ternary          UPDATE ternary          DELETE FROM ternary
  SELECT …                     SET    c = e1          WHERE a = e2
  FROM   …                     WHERE a = e2
```

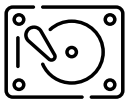
INSERT: evaluate query to construct new rows.

UPDATE, DELETE: query the table and identify the affected rows.

Modify table storage to reflect the row updates.[1]

[1] We still assume that the table has *no* associated index structures.

Using EXPLAIN on Q_4: INSERT



```
EXPLAIN VERBOSE
  INSERT INTO ternary(a,b,c)
    SELECT t.a, 'Han Solo', t.c
    FROM   ternary AS t;
```

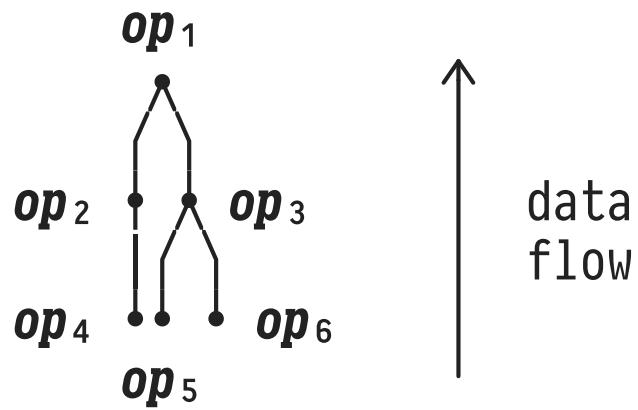
```
                                    QUERY PLAN
-------------------------------------------------------------------------------------
 Insert on public.ternary  (cost=0.00..20.00 rows=1000 width=44)
   ->  Seq Scan on public.ternary t  (cost=0.00..20.00 rows=1000 width=44)
         Output: t.a, 'Han Solo'::text, t.c
```

- **Seq Scan** scans table ternary to construct rows to be inserted, feeds 1000 rows into **Insert** for insertion.
- Width of inserted rows (over-)estimated to be 44 bytes = 4 (int) + 32 (text) + 8 (float) bytes.

Reading Complex EXPLAIN Outputs

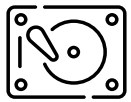
- EXPLAIN uses symbol -> and indentation to visualize larger, **tree-shaped** query evaluation plans:

```
QUERY PLAN
op1
  -> op2
       -> op4
  -> op3
       -> op5
       -> op6
```



- Read plans “inside out”, root op_1 delivers query result.

Using EXPLAIN on Q_4: UPDATE

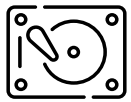


```
EXPLAIN VERBOSE
  UPDATE ternary AS t
    SET    c = -1
    WHERE t.a = 982;
```

```
                                    QUERY PLAN
-------------------------------------------------------------------------------------
 Update on public.ternary t  (cost=0.00..22.50 rows=1 width=51)
   ->  Seq Scan on public.ternary t  (cost=0.00..22.50 rows=1 width=51)
         Output: a, b, '-1'::double precision, ctid
         Filter: (t.a = 982)
```

- **Seq Scan** emits complete rows (only c was updated).
- Additionally feeds row ID (ctid) into **Update** to identify the affected row(s).

Using EXPLAIN on Q_4: DELETE

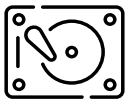


```
EXPLAIN VERBOSE
  DELETE FROM ternary AS t
  WHERE  t.a = 982;
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
Delete on public.ternary t  (cost=0.00..22.50 rows=1 width=6)
  ->  Seq Scan on public.ternary t  (cost=0.00..22.50 rows=1 width=6)
        Output: ctid
        Filter: (t.a = 982)
```

- **Seq Scan** returns affected row IDs (of 6 bytes each) only.
- We turn to **Filter** (makes scan skip non-qualifying rows) later in this course.

2 How do Row Updates Alter the Table Storage?

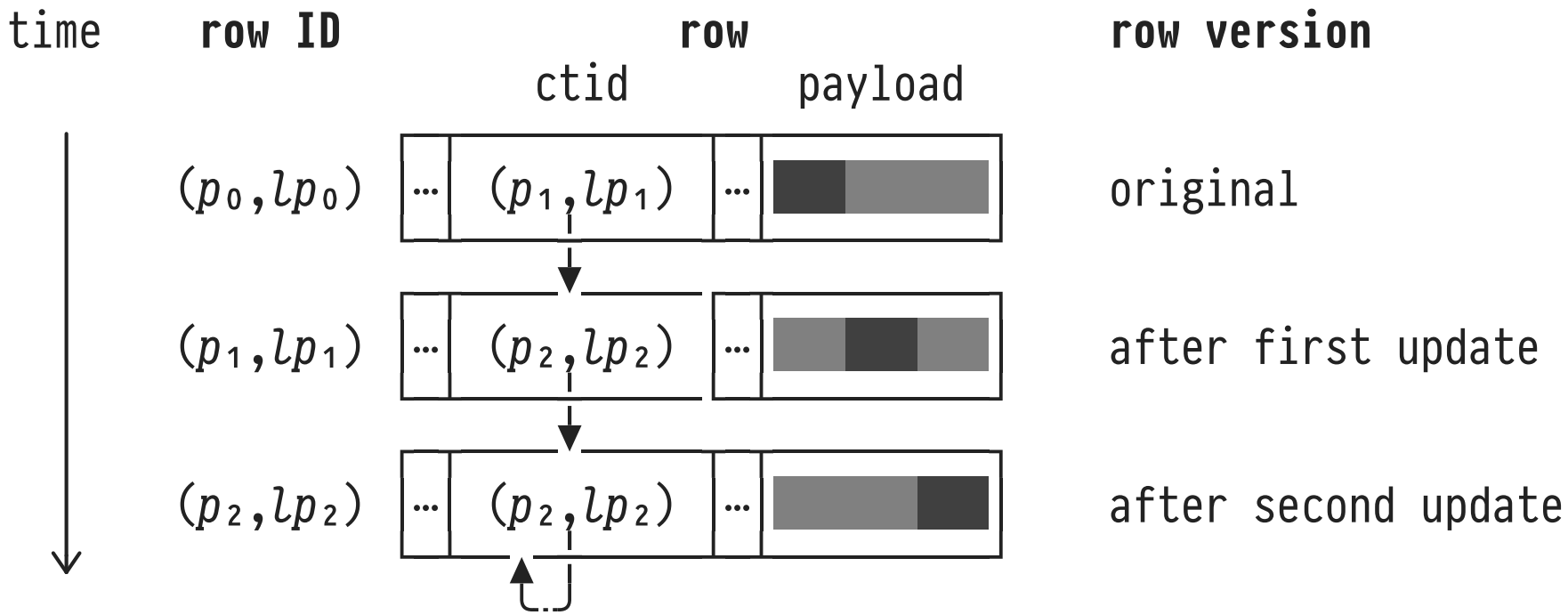


Let us take a closer look at how plan operator Update alters the target table's heap file pages. We find:[2]

- Rows are *not* updated in-place. A **new version of the row is created**—original and updated row co-exist.
- Any database user (query, application) sees exactly one version of any row at any time. Different users may see different row versions.
- A separate VACUUM ("garbage collection") step collects and removes old versions that cannot be seen by any user.

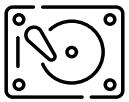
[2] This implementation of Update is typical for all DBMS that implement **Multi-Version Concurrency Control (MVCC).** We discuss MVCC later in this course.

Row Version Chains

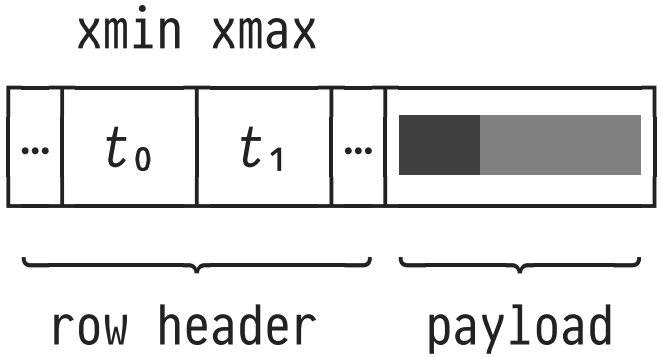


- Original and updated versions of a row form a **chain**, linked by the rows' IDs (held in row header field ctid).

Row Visibility and Timestamps



1. Each row carries two **timestamps**—xmin and xmax—that mark its first and last time of existence.
2. Each query/update is executed at some timestamp T which defines the rows that are **visible** to the operation:

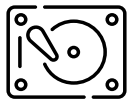


row is visible for any operation with timestamp $t_0 \leqslant T < t_1$

($t_1 = \infty$: row has not been updated yet)

- DBMS uses system-wide virtual timestamps (*transaction IDs*), see PostgreSQL built-in function txid_current().

Impact of Updates Beyond the Row's Page



- Updates on full pages may lead to row relocation across pages: versions then have row IDs (p_i, lp_i), (p_{i+1}, lp_{i+1}) where $p_i \neq p_{i+1}$.
  - Traversal of longer update chains may lead to I/O-costly “page hopping.”
  - $\Longrightarrow$ Perform VACUUM to collect inaccessible old versions. From outside page, point to most recent row directly.

- PostgreSQL optimizes for the good-natured case where $p_i = p_{i+1}$ and indexed row fields have *not* been changed.
  - Such **heap-only tuple (HOT) updates** have page-internal impact only, no maintenance outside page required.

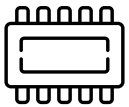
3 ⋮ Q_4 — Row Update

`INSERT INTO` ternary `VALUES` (…,…,…)	`UPDATE` ternary `SET` c = e_1 `WHERE` a = e_2	`DELETE FROM` ternary `WHERE` a = e_2

UPDATE: affects updated column(s) only.
INSERT, DELETE: operate on full rows, all column BATs of table ternary will be affected.

- MonetDB uses user-specific **Δ tables ("delta tables")** to represent changes. Column BATs are *not* modified immediately. Global visibility of changes is delayed.

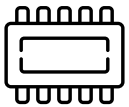
Using EXPLAIN on Q_4: DELETE



```
sql> EXPLAIN DELETE FROM ternary WHERE a = 981;

 ⋮
ternary     :bat[:oid] := sql.tid(sql, "sys", "ternary");
a0          :bat[:int] := sql.bind(sql, "sys", "ternary", "a", 0:int);
deleted_rows:bat[:oid] := algebra.thetaselect(a0, ternary, 981:int, "==");
sql_delete             := sql.delete(sql, "sys", "ternary", deleted_rows);
 ⋮
```

- algebra.thetaselect(b_1,b_2,v,θ) returns the oids of those rows r in algebra.projection(b_2,b_1), for which predicate tail(r) θ v holds.
- sql.delete(…) modifies the BAT of currently visible rows (obtainable via sql.tid(…)) for table ternary.
  ⚠ However, no column BAT is changed yet.

Using EXPLAIN on Q_4: INSERT

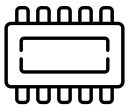


```
sql> EXPLAIN INSERT INTO ternary(a,b,c) VALUES (1001, 'Han Solo', -2);

 ⋮
sql_append := sql.append(sql, "sys", "ternary", "a", 1001:int);
sql_append := sql.append(sql, "sys", "ternary", "b", "Han Solo");
sql_append := sql.append(sql, "sys", "ternary", "c", -2:dbl);
 ⋮
```

- For ternary(a,b,c), a row insert translates into three individual operations on the column BATs.
- sql.append(…) saves the inserted value in the **Δ^i table** associated with each column BAT.

  ⚠️ The column BATs do not change yet.

Using EXPLAIN on Q_4: UPDATE



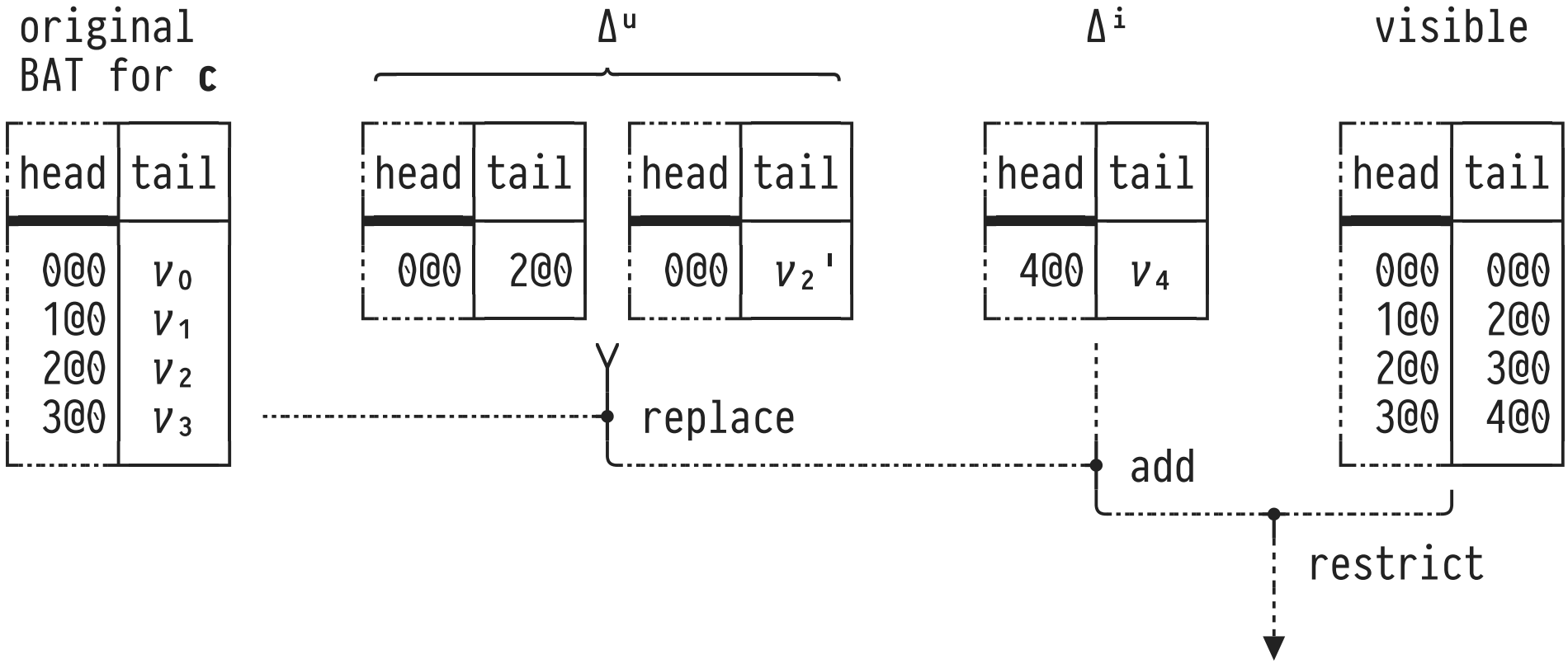
```
sql> EXPLAIN UPDATE ternary SET c = -1 WHERE a = 982;

 ⋮
ternary       :bat[:oid] := sql.tid(sql, "sys", "ternary");
a0            :bat[:int] := sql.bind(sql, "sys", "ternary", "a", 0:int);
updated_rows  :bat[:oid] := algebra.thetaselect(a0, ternary, 982:int, "==");
updated_rows_a:bat[:int] := algebra.projection(updated_rows, a0);
updated_c     :bat[:dbl] := algebra.project(updated_rows_a, -1:dbl);
sql_update               := sql.update(sql, "sys", "ternary", "c",
                                       updated_rows, updated_c);
 ⋮
```

- BATs updated_rows and updated_c contain oids and c values of the changed rows.[3]
- sql.update(…) saves these changes in the **Δ^u table** for the BAT of column c. ⚠ The column BAT is not changed yet.

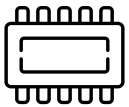
[3] algebra.project(*b*,*v*) returns *b* with all tail values set to *v*.

Δ and Visibility Tables

In column c, update v_2 to v_2', insert v_4, delete v_1:



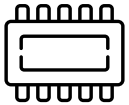
Applying Changes on Demand



- When a query needs to see the changes made to column `c`, apply all changes accumulated in the `c`'s Δ tables:
  1. Load—yet unmodified—column BAT for `c`.
  2. Read `c`'s Δ^{u} table and perform value replacements (via `bat.replace(…)`).
  3. Read `c`'s Δ^{i} table and perform value inserts (via `bat.append(…)`).
  4. Restrict `c` to currently visible rows (via `algebra.projection(…)`).

- Make changes permanent only once we want them to be seen globally by all users (*COMMIT* → transaction management).

Delay Change Propagation: Disable *AUTO COMMIT*



- To experiment with Δ-based change management, disable MonetDB's default *auto commit* behavior (`mclient` option `--autocommit` or command `\a` to turn *auto commit* off):

```
DELETE FROM ternary WHERE a = 981;
INSERT INTO ternary(a,b,c) VALUES (1001, 'Han Solo', -2);
UPDATE ternary SET c = -1 WHERE a = 982;
```

- Without *auto commit*, changes are still pending at this point. Thus:

```
SELECT t.c              ⎫  reads Δᵘ, Δⁱ (+ visibility table)
FROM   ternary AS t;    ⎭  to reflect changes on t.c
```