Lecture 4 - Berkeley History

Papers for today

- The POSTGRES Next-Generation Database System (1991) -
  - written close to when the Postgres team commercialized it.
  - full description of Postgres vision and architecture
- The Design of the Postgres Storage System (1987) -
  - a description of their storage manager using version and no-overwrite,
  - rather than WAL
- Looking Back at Postgres (1999) - retrospective on impact of Postgres
- Reason we're reading it is because Postgres did a "lot" of things - and had huge impact across a range of topics
  - Extensibility (types, functions, operators)
    - they call this "object management"
    - vast majority of this is done through existing system catalogs (an another example of dogfooding)
  - Rules system
    - they call this "knowledge management"
  - Multi-versioned storage
    - no overwrite storage - departure from full WAL
- Q: Thoughts on the paper?

CACM paper

Background

- Written in 1991; Postgres under development since 1986. Earliest papers:
  - The design of POSTGRES, SIGMOD'86
  - The POSTGRES Data Model, VLDB'87
- Wild that you can still run a piece of software 35 years later!

Design Philosophy

- *Orientation towards a set-oriented-query language*
  - POSTQUEL (PQ)
    - SQL support arrived with **Postgres95 (1994)**, and the system was renamed **PostgreSQL in 1996**
  - Some vestiges of CODASYL still remain (but deemphasized)
  - Declarative, set-oriented querying was central...
  - But: enable a "fast path"
    - Essentially allow functions to be registered and executed;
    - Can mix imperative code, PQ, as well as low-level functions ("get next record")
- *Multilingual access*
  - Departure from OODBs at the time that persist all program objects and were tied to a PL ->
    - here, you push PL constructs into a database
    - and instead use any PL to access it
    - this separation is now standard
- *a small number of concepts*
  - classes, inheritance, types, functions
    - of these inheritance is perhaps least emphasized in modern DBs, most of the others are folded into modern DBs in some way

- Honestly feels like a lot - but not compared to the complexity of systems at a time

Data Model

Classes and Inheritance

- a class ≡ relation; instances ≡ records
- Instances have OIDs - identifiers that can't be modified
- Inheritance:
  - `create EMP (name = c12, salary = float, age = int)`
  - `create SALESMAN (quota = float) inherits EMP`
- Can also inherit from multiple paths
- Different types of "classes"
  - Base/real - just a relation
  - Derived/virtual - just a view
  - Version - will see this later

Types

- Three types of types (ha)
  - Base types (which can also be extended)
  - Arrays of base types
  - Composite types
- Base types
  - The usual, but you can also extend to *Abstract Data Types*
  - Users can declare, they just need to provide a way to serialize it into strings
    - Though this may be costly/inefficient
  - `create DEPT (dname = c10, floorspace = polygon, mailstop = point)``
  - `replace DEPT (mailstop = "(10, 10)") where DEPT.dname = "shoe"`
  - Early form of extensible ADTs in modern DBs
- Arrays of base types
  - `create EMP (name = c10, salary = float[12])`
  - `retrieve (EMP.name) where EMP.salary[4] = 1000`
  - Most databases now support array operators
- Complex objects
  - Any class is also a type; and when you add that class as an attribute to another class, it becomes a container for zero or more instances of that class (weird, i know)
  - `create EMP (name = c10, salary = float[12], manager = EMP, coworkers = EMP)`
  - Both `manager` and `coworkers` end up being arbitrary sized lists of instances, essentially - afaict, no way of constraining to be a single instance
  - Q: how would we implement this in standard relational model?
  - Q: we learned about pitfalls of CODASYL - no nesting, no pointers - isn't this bad??
    - Yes, partially - even though nesting is present, it is not tied to physical implementation
  - Second flavor of complex object is a `set` - collection of arbitrary types
    - This feels just like JSON to me
  - `retrieve (EMP.manager.age) where EMP.name = "Joe"`
    - Dot notation, much like you would use in XPath, or in JSON in SQL

Functions

- Three types of functions: C functions, operators, PQ functions
- C UDFunctions:
  - arguments are base/composite types
  - `retrieve (DEPT.dname) where area (DEPT.floorspace) > 500`
  - `retrieve (EMP.name) where overpaid (EMP)`
  - Note: no optimization of such UDFs
    - still a difficult problem
- Operators:
  - `retrieve (DEPT.dname) where DEPT.floorspace AGT "(0,0), (1,1), (0,2)"`
  - Can be associated with access methods
    - R-Trees, KD-Trees, Joe later added GiST as part of his PhD thesis
    - Modern impact: Postgres uses GiST as a foundation (e.g., geometric, ranges); alongside other extensible index families (GIN, SP-GiST, BRIN).
- PQ functions:
  - Any collection of commands in PQ can be used to package a function
  - `define function high-pay returns EMP as retrieve (EMP.all) where EMP.salary > 50000`
  - Can use in PQ queries as well (essentially a subquery)
    - `append to EMP (name = "Sam", salary = 1000, manager = mgr-lookup ("shoe")`
  - Or directly as well - this is that *fast path*
    - Generally user can directly call the parser, optimizer, executor, access methods, buffer manager, etc.
    - *it provides direct access to specific functions without checking the validity of parameters.* - therefore more efficient

Query Language Extensions

- Nested queries
- Transitive closure
  These two are more interesting:
- Time travel
  - `retrieve (EMP.salary) from EMP [T] where EMP.name = "Sam"`
- Inheritance
  - `retrieve (E.name) from E in EMP* where E.age > 40`
  - retrieves from all subclasses of `EMP` too

Rules System

- Goal - go beyond referential integrity and further push application logic and constraints to the database
  - *"referential integrity [7], which is merely a simple-minded collection of rules."*
- PG's rule system was very bold, encompassing:
  - view maintenance
  - triggers
  - integrity constraints
  - referential integrity
  - version management
- `on new EMP.salary where EMP.name = "Fred" then do replace E (salary = new.salary) from E in EMP where E.name = "Joe"`
- or equivalently `on retrieve to EMP.salary where EMP.name = "Joe" then do instead retrieve (EMP.salary) where EMP.name = "Fred"`
  - triggers on a read! Very unusual

- Former approach would work better if few updates of Fred, more reads of Joe latter if few reads of Joe, more updates of Fred.
    - not ideal that user has to pick
- Two main implementation methods:
    - Tuple level processing
    - Query rewrite
- Tuple level processing is very unusual
    - In example 1 above, marker placed on salary attribute of Fred's instance
        - If update touches attribute, then rule is called
        - Many corner cases; e.g.,
            - Fred's name is changed then Marker is dropped
            - Joe exists before Fred - marker added for first Fred
    - Performance issues with too many markers - escalate and add an enclosing marker
    - This only makes sense if there are specific rules for a small # of records
    - If rule touches most of table, then running this subroutine per query is overkill
- Query rewrite
    - Works better for queries that touch more
    - But rewriting while preserving semantics is nontrivial
    - Generally there is confusion around semantics with multiple instances - tuple level and query rewrite approach will do different things
- Modern triggers in practice go for more straightforward implementations: ECA - primarily for updates
- Rule systems for implementing views
    - `define view TOY-EMP (EMP.all) where EMP.dept = "toy"` implemented as:
    - `on retrieve to TOY-EMP then do instead retrieve (EMP.all) where EMP.dept = "toy"`
    - What is known at the time: updating to views is only done when it is unambiguous
        - Q: what is an example of a query where the view update is not ambiguous? ambiguous?
    - Postgres allows users to implement their own view update logic
- Rule systems for implementing versions:
    - `create version my-EMP from EMP`
    - Basically a fork, where updates can be done to this version without impacting the original
    - implemented as:
        - two classes:
            - `EMP-MINUS (deleted-OlD)`
            - `EMP-PLUS (all-fields-in EMP, replaced-OID)`
        - rewritten queries:
        - `on retrieve to my-EMP then do instead
        - `retrieve (EMP-PLUS.all)
        - `retrieve (EMP.all) where EMP.OID not in {EMP-PLUS.replaced-OID} and EMP.OID not in {EMP-MINUS.deleted-OID}`

Storage Engine

- Also the focus of the second paper!
- wanted "*to do something different*" - thought WAL was overkill
- WAL enables **STEAL + NO-FORCE** buffer management:
    - pages can be written before commit (steal)
    - pages don’t have to be forced at commit (no-force)
    - instead, force the log (sequential I/O) to guarantee durability

    - ARIES wasn't yet known (1992) - but mostly it focused on "completing" the story when there are crashes during recovery etc.
- Fundamentally different approach for recovery
    - a "no overwrite" storage manager
    - so: purpose served by WAL is done in regular storage
    - WAL now replaced by a log of status of each transaction (2 bits)
    - tuple versions + transaction status log is sufficient!
- Benefit 1: instantaneous abort and crash recovery
    - No need to undo updates of uncommitted txns, redo updates of committed txns
    - Everything is reflected in the regular data - we just need to abort the inprogress txns, and not change the regular data
- Benefit 2: time travel is possible.
    - Queries for data "as of" time T
    - Theoretically one could try to do so with the log but the log would need to be augmented.
- Downside:
    - All pages for a committed transaction must be forced to disk
        - to ensure durability of committed transaction
        - this can involve many random writes;
        - contrast to flushing a log, which is sequential & cheaper
        - generally this would work well with a more "stable" memory
        - Modern context:
            - Lots of hope for NVRAM - but never materialized
            - most prominent recent commercial attempt—**Intel Optane persistent memory**—was discontinued in 2022
    - "All data" mixed in; if there are lots of aborts or even lots of updates, there's lots of data to wade through
- Postgres can also migrate from one long term storage format (disk) to archival (tape)

Additional Details from second paper

- Lots of "back of envelope" math to justify the fact that transaction statuses could be stored in main memory
    - But now not a big deal - memory is much cheaper and plentiful!
- Per record, store the following:
    - OID a system-assigned unique record identifier
    - Xmin the transaction identifier of the interaction inserting the record
    - Tmin the commit time of Xmin (the time at which the record became valid)
    - Cmin the command identifier of the interaction inserting the record
    - (similarly, Xmax, Tmax, Cmax)
    - PTR a forward pointer
- Basically Xmax, Tmax, Cmax will be NULL while the record version is still valid, and then get updated when the record is updated or deleted.
- For new versions of a record, can additionally only restrict to storing delta (only attributes that change)
- A record is valid at time T if the following is true:
    - Tmin < T and Xmin is a committed transaction and either:
        - Xmax is not a committed transaction or
        - Xmax is null or
        - Tmax > T
- Uses locking

Some perspectives from Joe's paper

- Abstract Data Types and Functions:
  - All major vendors have support for hierarchical data via JSON/XML, using very similar path notation to what Postquel did
  - All major vendors support UDFs
    - One could argue that MR revolution was a need to support UDFs + parallelism -> now certainly looks that way
  - Extensible data types
    - Postgres now supports GIN (search for unstructured data), GiST
  - OR won out against OO - additional OO functionality came in the form of ORMs - which were different from both
  - UDF query optimization
    - Still really hard
    - Q: Suppose we have a user defined function predicate - should we still push it down?
    - Joe worked on this problem as part of his thesis
- Active databases
  - Triggers are part of the database standard
  - But are really hard to enforce in practice
  - See similar ideas in materialized view maintenance, CEP, streaming
- Version-centric storage
  - Realization: current state is simply a view over versions
    - Modern incarnations: LSM trees in KV stores
  - Unfortunately never excelled in performance, versioning and time-travel was replaced by WAL in 2001; see announcement (time travel was removed even earlier, 1998)
  - But - within postgres, multi-versioning ended up being useful to support MVCC (added in 1999) - and snapshot isolation - more on that at some future point
- Open Source
  - SQL porting: 1995 - called PostgreSQL after that
  - Many startups, often around parallel postgres
  - Andy Pavlo's 2026 seminar series Postgres vs. the world

    > "Postgres was designed for extensibility, and that design was sound. With extensibility as an architectural core, it is possible to be creative and stop worrying so much about discipline: you can try many extensions and let the strong succeed."

> Another lesson is that a broad focus—“one size fits many”—can be a winning approach for both research and practice.