Lecture 2 - Data Models

What's happening today

- Main paper: "What goes around comes around" (2005), Stonebraker (mainly), according to Joe.
- Second paper: Revisiting that via "...and around" (2025), Stonebraker and Pavlo
- Both papers are very opinionated (you can think of them as "thought pieces")
  - **do not** believe everything as written!
  - heavy bias towards systems the authors have written!
- Third (optional) paper: "A relational model of data for large shared data banks"
  - yes, this is the Codd paper
- Goal - revisit explorations of various data models over time
  - 1960s--2005; 2005--2025
  - what were the lessons from these eras - on the data model side
- Let's start by talking about the first paper, which was part of the Redbook
  - Main motivation of that paper was that all data models were revisiting arguments in the hierarchical/network data model and the relational data model
- First, some concepts from the Codd paper:
  - Physical data independence & logical data independence
  - Physical: applications continue to work as physical organization changes
  - Logical: applications continue to work as logical organization (schema) changes

Various data model eras

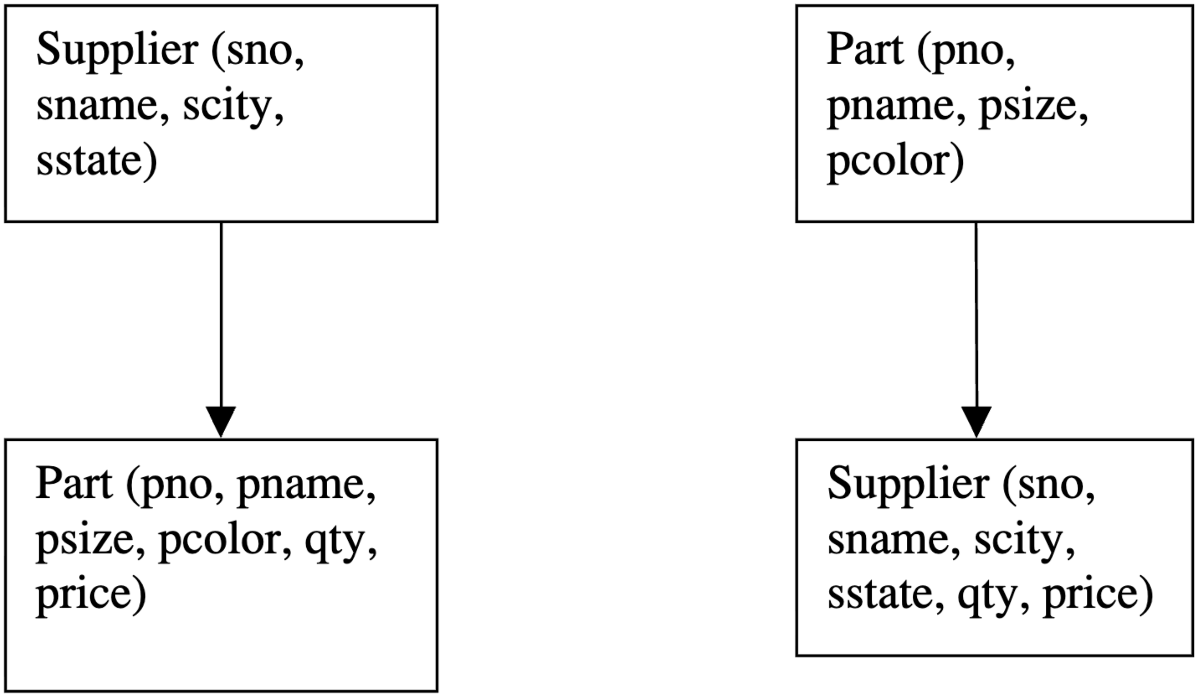
- Hierarchical (IMS): late 1960’s and 1970’s
- Network (CODASYL): 1970’s
- Relational: 1970’s and early 1980’s
- Entity-Relationship: 1970’s
- Extended Relational: 1980’s
- Semantic: late 1970’s and 1980’s
- Object-oriented: late 1980’s and early 1990’s
- Object-relational: late 1980’s and early 1990’s

Example setup

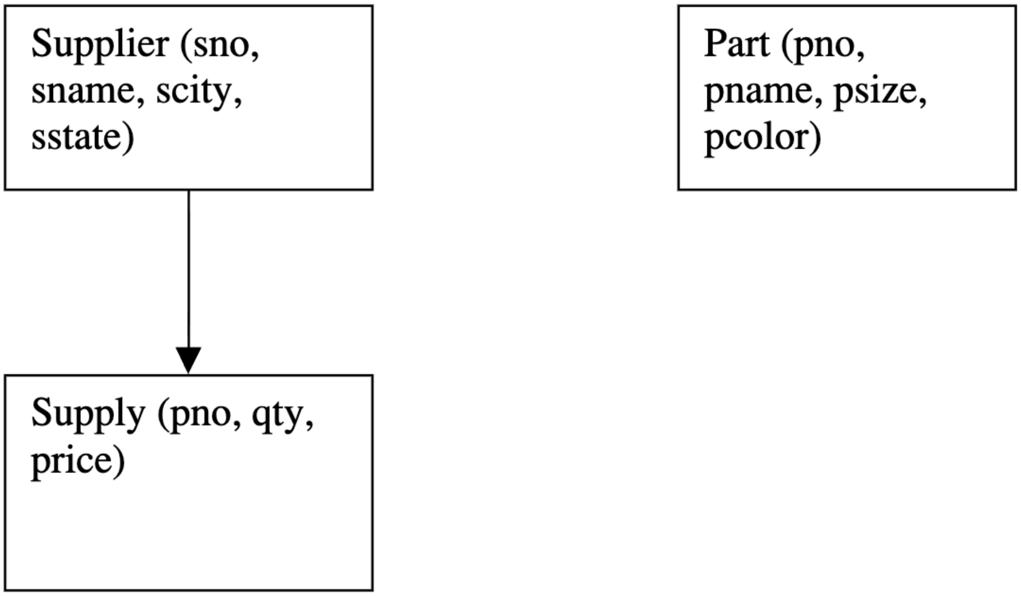
- Supplier (sno, sname, scity, sstate)
- Part (pno, pname, psize, pcolor)
- Supply (sno, pno, qty, price)

IMS (hierarchical) era (1960s)

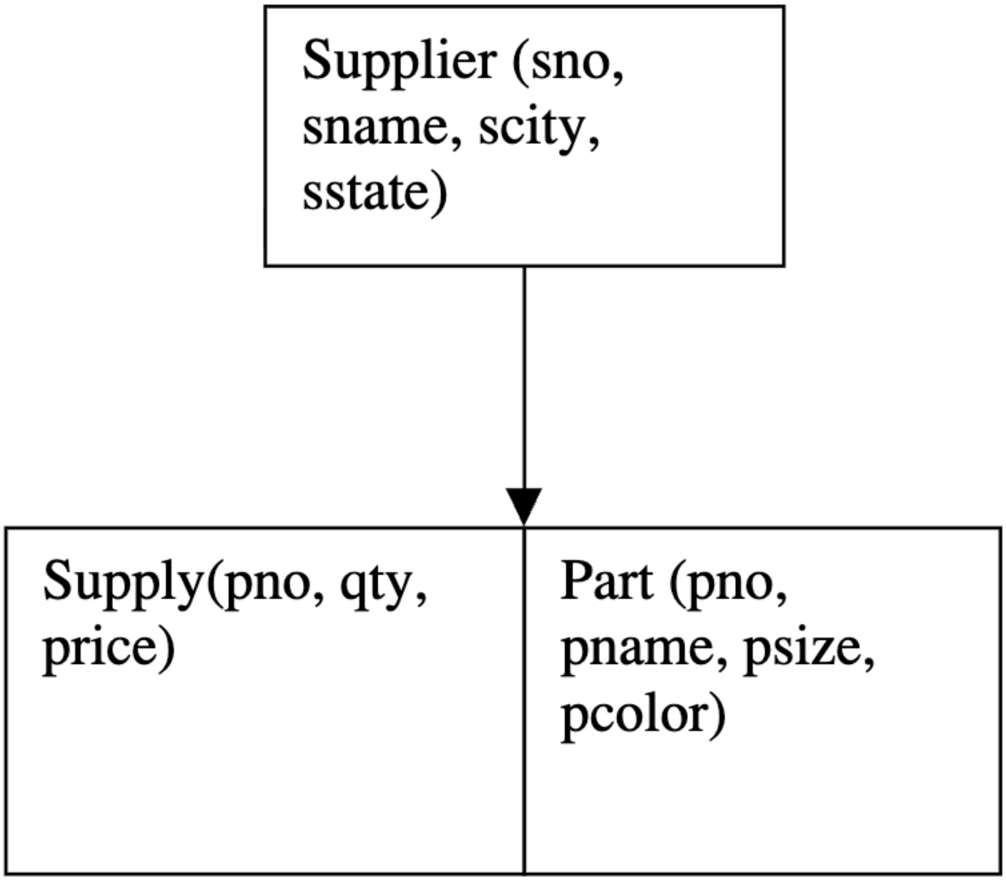
- Data Model:
  - Records have fields and a key (a set of fields)
  - Record types (schemas) are hierarchical:
    - arranged in a rooted ordered (will see why this matters) tree, where each record type has a parent record type
- One approach - where Supply gets "squished" into child node or alternative


Supplier (sno, sname, scity, sstate)
Part (pno, pname, psize, pcolor, qty, price)
Part (pno, pname, psize, pcolor)
Supplier (sno, sname, scity, sstate, qty, price)

- Main issues with this approach:
  - Redundancy:
    - On LHS, part info is repeated for each supplier who supplies the part
    - Redundancy is not great because it raises the danger of inconsistencies
  - Existence: A part can't exist without a supplier (same for other)
  - Some queries are easier than others - depending on nesting order
- Query Execution:
  - Record-at-a-time
  - Retrieval of records done by using hierarchical sequence key: a concatenation of keys along the path to the root
  - Users need to figure out where to start and how to traverse
  - Expectation that traversal happens depth-first, left-to-right
    - Essentially following physical pointers, or sequential scan
- On Data Independence:
  - Violates physical data independence because the program could expect that the data is laid out in a certain way, e.g., pointers from parents to children.
  - Some amount of logical data independence because you can extend tree by adding additional roots
- To avoid redundancy, one can do the following:


Supplier (sno, sname, scity, sstate)
Part (pno, pname, psize, pcolor)
Supply (pno, qty, price)

- But the challenge is that combining information across "trees" is impossible
  - so mechanism to do so is to physically stitch things together through pointers,
  - from the programmer's standpoint this looks like this, even though there's only one copy of each part, which is reachable from each supply

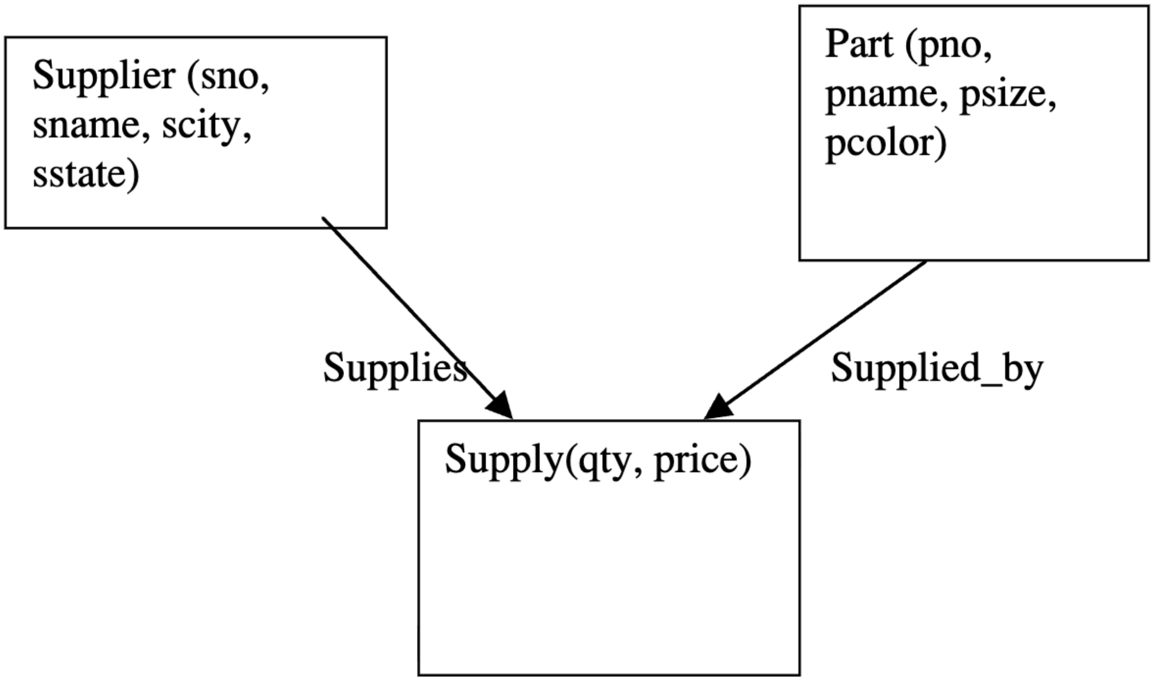

Supplier (sno, sname, scity, sstate)
Supply(pno, qty, price)
Part (pno, pname, psize, pcolor)

- Overall, tree structured data models have various downsides

    - don't provide any real physical or logical data independence.
        - can't really reorganize the data
    - record-at-a-time imperative query specification approach requires users to make tradeoffs:
        - should they start at the root and traverse down following pointers, or:
        - jump to a node and then walk down, etc.
        - no real query optimization across these alternatives - all manual
    - the representations are restrictive - combining information across entities isn't possible unless they are hierarchically organized within each other
    - schema defines both logical relationships and physical navigation paths
        - schema equals access methods

CODASYL (network) era (1970s)

- This got Charles Bachmann the Turing award
- Main issue with IMS is that each record type has a single parent
    - Record types as in IMS, but now a network rather than a tree


Supplier (sno, sname, scity, sstate)
Part (pno, pname, psize, pcolor)
Supplies
Supplied_by
Supply(qty, price)

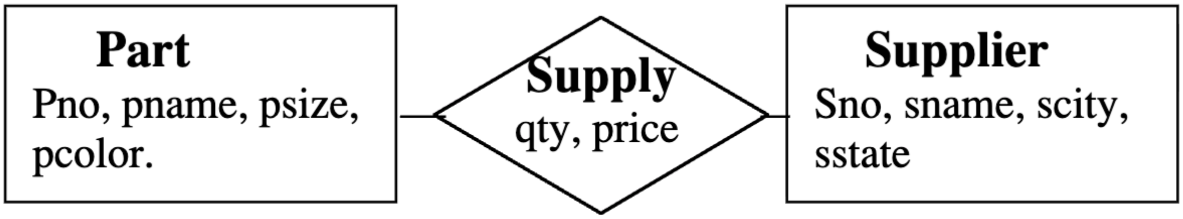
    -
- Data Model
    - Basically, data model: record types + sets
        - Supplies and Supplied_by edges (these are called "sets") indicate that for each parent, there is a relationship with 0 or more children.
    - Each set, however, can only support a two-way relationship, not multi-way
        - Eg sale - salesperson, customer, item
- Query Execution:
    - Still record-at-a-time` `` `Find Supplier (SNO = 16) Until no-more { Find next Supply record in Supplies Find owner Part record in Supplied_by Get current record -check for red- } ``
    - Essentially following pointers between parents and children
    - But now, even more complicated, since one needs to keep track of what all one has visited in the context of each individual loop and also globally
        - In IMS, the programmer needs to only keep track of where they are currently, and their parent
        - Here, one may need to go down, sideways, up, and down again ...

        - Essentially, a user is implementing a form of complicated directed nested loop, manually - keeping track of where they are for each step
            - Bachmann: "navigating in hyperspace"
- Independence:
    - Because of these pointers, no physical data independence
    - Similarly, programs require the network schema, so no logical data independence

Relational Era (1970s and 1980s)

- From paper "*the driver ... was the fact that IMS programmers were spending large amounts of time doing maintenance on IMS applications, when logical or physical changes occurred.*"
- Three maxims (from paper)
    - *Store the data in a simple, flat data structure (tables)*
        - flexible also to represent multi-way relationships
    - *Access it through a high level set-at-a-time DML*
        - no imperative nested loops to traverse data
    - *No need for a physical storage proposal*
        - database system decides how to store
        - this means no pointers, no expectation that records can be located by following certain paths
- Truly declarative - users are not manually optimizing their programs, systems can figure it out
- Led to "the great debate" between Codd and Bachmann at SIGMOD'74
    - Codd: CODASYL is too complex, too little physical independence, nested-loop based programming is hard to optimize
    - Bachmann: Can never be efficient
- By '76, INGRES and System R had shown that efficient implementations were possible; relational model came with physical data independence anyway.
    - INGRES also introduced the notion of views, adding to logical data independence
- By '84 IBM prioritized relational systems - ending the "great debate"

ER Model (1970s)

- Originally proposed by Peter Chen as an alternative to relational, CODASYL, hierarchical data models - and trying to unify them
- Data model
    - entities + relationships, each with attributes
    - 

    - One way it was "better" than relational is that it made relationships a first-class citizen
- Did not actually have a proper query execution proposal
- Eventual use-case
    - DB design (translate into relational), rather as a serious data model proposal - no serious system supported it
    - Alongside, DB theory folks developed theories of normalization (including Codd, who introduced normalization in the first place)
        - 2NF, 3NF, BCNF, 4NF

    - Stonebraker disses on normalization "*Functional dependencies are too difficult for mere mortals to understand*."
    - But still very useful to understand where redundancy can occur in a schema
        - Even if we choose to not do anything about it

Relational Extensions (early to mid 1980s), Semantic data models (1980s)

- Relational ++
    - Supporting set oriented attributes:
        - e.g., Part(pno, pname, available_colors = {red, blue, green})
        - Avoids joins but breaks atomic attribute property
    - Foreign keys via tuple pointers
        - e.g., Supply(PT: Part, SR: Supplier, qty, price), PT, SR - pointers to tuples in those tables
        - Like CODASYL's pointers
    - Sub-classing (specialization/generalization)
        - Didn't really catch on until OO languages took off
- Semantic data models
    - Added in nested records
    - Multiple inheritance
- Why didn't this take off?
    - Mainly this wasn't where the pain was: RM could handle it
    - Not enough added functionality, most of the pain was in performance

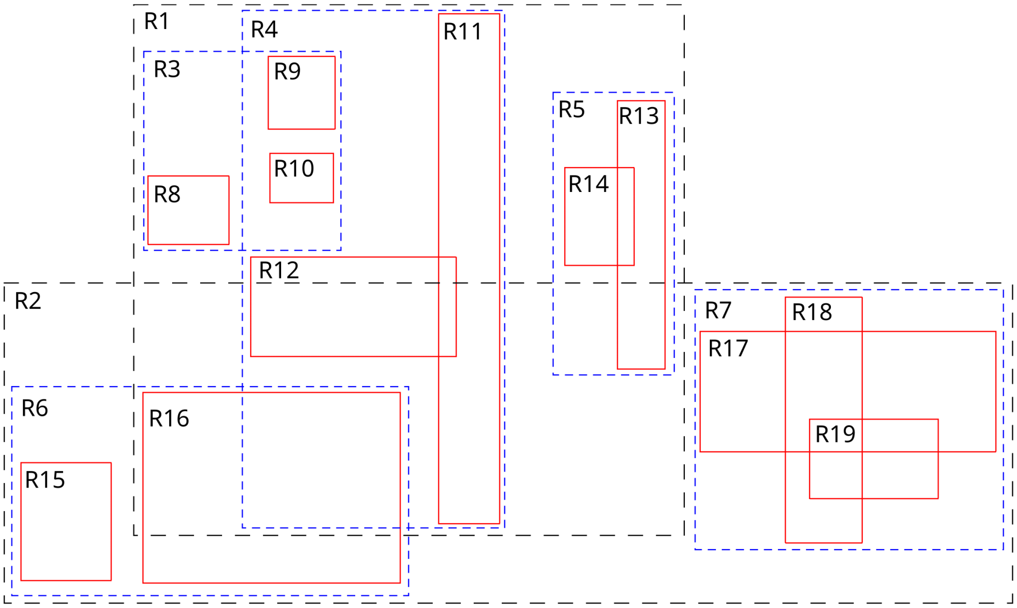
OODB (1980s to 1990s)

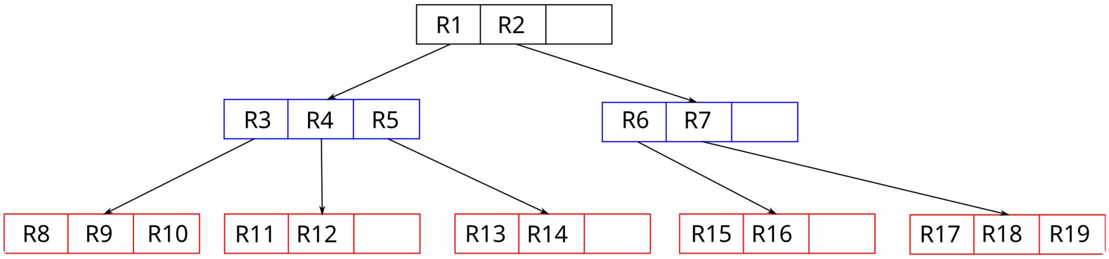
- C++ on the rise, with object orientation
- Main problem they wanted to solve:
    - impedance mismatch between programming languages and databases
    - also wanted support for complex objects (e.g., CAD)
- Made PL objects = database objects
    - Declare a variable -> gets persisted
        - Persistent Part p;
          p.color = "red"; // persisted
    - Early versions of ORMs
- Why didn't this take off?
    - People wanted to not be tied to a programming language
    - Needed buy in from PL folks
    - Ends up being network/hierarchical data model again

OR (1980s and 1990s)

- Major motivation was GIS (map data)
    - Want to be able to support range queries (lat long), without having separate indexes for lat and long - as this is pretty inefficient
    - Need for complex data types and support for indexes on complex data types
    - Want all of this as part of SQL
- Core idea: make DB extensible
    - User defined types, operators, functions, access methods
        - e.g., intersection operator "WHERE point spatially_in "x0, x1, y0, y1""
        - internally calls UDF - user defined function

    - Basically database engine needs more flexibility to support these complex data types
- Outcomes:
    - Postgres was the major successful system of this era
    - New indexes become commonplace
        - R-tree, Quad-Tree
        - Detour to present R-trees
            - Height balanced tree (like B-tree)
            - Each node has between m to M pointers
            - With each pointer, we store MBR: minimum bounding rectangle
            - For search, we will go down each MBR that overlaps with rectangle (maybe multiple search paths in worst case)
            - For insertion, go down MBR that will expand the least at each step
                - Find leaf where the object fits, and if it does, great
                - If not, split, and then propagate split upwards (just like B+tree)





    - Another innovation: stored procedures
        - Define multi-SQL queries that can be treated as a unit - essentially a predefined UDF
- However, the extensibility, while really helpful for having relational databases adapt to new requirements, wasn't the major win.
    - paper: "*the major win in Postgres was UDTs and UDFs; the OO constructs were fairly easy and fairly efficient to simulate on conventional relational systems*"
    - UDFs bring code to data, as opposed to data to code - reducing data transfer
        - specialized programs that can't be expressed in SQL
    - To this day, UDFs are essential to support important "external" primitives; think any libraries, ML, LLMs, specialized functions for each vertical...

    - Another way to think of this is that it was an attempt towards adding in imperative PL constructs into a declarative system
    - from paper: "*The major benefits of OR is two-fold: putting code in the data base (and thereby blurring the distinction between code and data) and user-defined access methods.*"

Semi-structured data (2000s)

- Two main ideas
    - Schema-last, instead self-describing format
    - Nested, hierarchical model
- Canonical setup
    - XML (now JSON)
    - `<person><name>Mike Stonebraker</name><age>85</age></person>`
    - Set of attributes can be different; could be nested
    - Paper walks through various use cases, arriving at one compelling one: resumes
    - Moving it closer to relational data - XML Schema, DTDs
        - Paper's prediction "*XML Schema will fail because of excessive complexity*"
- Parallels to IMS and CODASYL
    - Hierarchical like IMS, with same issues of redundancy etc.
    - Some CODASYL features too
    - Set based attributes, like SDM
    - From paper: "*The current proposal is now a superset of the union of all previous proposals. I.e. we have navigated a full circle.*"
- Some deviation from IMS/CODASYL
    - instead of users looping through, and using pointers
    - Instead use set-at-a-time query language, Xquery (or XPath)
- Implementation-wise: Challenges in supporting even a simple variant: moving from atomic type to union type, is difficult
    - what would index look like - require hybrid index
    - operators require custom implementations - need to account for each type interacting with each type of the other "side"
- What has stood the test of time:
    - semi-structured data is an excellent interchange format

Main takeaways from paper

- Only two good ideas from the entire set of data model proposals:
    - UDFs
    - Schema last - but only for data interchange

(Didn't actually cover the following in class)

... and around

- Paper from 2024 - recaps a number of advances from the 2000s and 2010s
    - Many of these are not really data models but different system architectures
- Their conclusion is similar "*that the relational model with an extendable type system (i.e., object-relational) has dominated all comers, and nothing else has succeeded in the marketplace.*"
    - Generally true but there are specialized systems that work really well in certain settings
    - What is also true that the NoSQL movement of the late 2000s has come full circle to embrace RM and SQL

    - Mostly rejected RM as being not good enough performance-wise

Various new proposals

1. Map Reduce Systems
2. Key-value Stores
3. Document Databases
4. Wide-Column
5. Text Search Engines
6. Array Databases
7. Vector Databases, and
8. Graph Databases.

1. Map Reduce

- Parallel programming framework, based on low-level primitives map and reduce
    - One way they describe it is as: `SELECT map() FROM crawl_table GROUP BY reduce()` - but `map()` is done first
- Not really a data model proposal, more a programming abstraction proposal but suffers from similar issues
    - Moved away from declarative to require the programmer to specify the low-level imperative steps required to perform batch tasks

2. KV store

- Only store KV pairs, no schema
- Generally fine if keys and values are flat and any processing required for the values can be done at app layer
- Downsides:
    - App layer will need to parse any complex V into individual fields, and also there are no secondary indexes.
    - No way to do joins or combine information across multiple KV pairs.
- But, there are still some popular (transactional) KV stores - thanks to their speed - DynamoDB is one
- Sometimes people write new DBs using existing KV store as storage engine
- From paper *"Many have either matured into RM systems or are only used for specific problems"*
- Most common examples: Redis, DynamoDb

3. Document DBs

- JSON data model - flexibility, performance (nesting helps keep info together, no join necessary)
- Different API that didn't involve SQL, and mostly used either a MR interface, or a bit closer to relational operators
- Very complicated to work with, optimization is hard for the system - onus is on user to write optimized "plans"
- Also same issue as hierarchical/nested data models - redundancies, limited physical data independence
- Likely will diminish over time - with JSON becoming a type in traditional RDBMSs
- Examples: MongoDB, Couchbase

4. Wide Column DBs

- A canonical implementation here was Google’s BigTable in 2004, which got abandoned in 2010s, replaced by Spanner, fully relational
- Main idea was to have different subsets of rows with different subsets of columns
- Complicated to work with, similar issues as the union typing issues mentioned earlier

5. Text search engines

- Supporting text search by intersecting posting lists
  - essentially boolean conjunctive/disjunctive queries
- Specialized systems like Elastic, Solr, Lucene, ...
- But could also use a text index on a text column in a DBMS
  - May not be as performant but more convenient

6. Array databases

- Vectors, matrices, tensors,
  - Data where the "location" is paramount
- Main use-case is scientific, e.g., astronomy data
- Could also be used for LA and therefore ML but is an imperfect fit because usual expectation is that the arrays are sparse
- Some specialized systems, like SciDB
- But SQL now has array capabilities as of 1999

7. Vector databases

- A form of dense arrays, but storing embeddings from DL
- Main use-case is embedding similarity; k-NN
- Specialized systems Pinecone, Milvus, and Weaviate
- Most DBs have added vector capabilities as well
- Text search engines have also added them

8. Graph databases

- Most compelling use-case: primitives for graph traversal
- Performance may be better for RM with
  - Node (node_id, node_data)
  - Edge (node_id_1, node_id_2, edge_data)
- But ergonomics of PG or triple store languages are definitely better.
- Most prominent example: neo4j and Cypher

Big takeaway:

- RM + extensions (UDTs, UDFs) are enough to handle the vast majority of use-cases
- Most else is about special use cases (eg search) or applications (eg science)