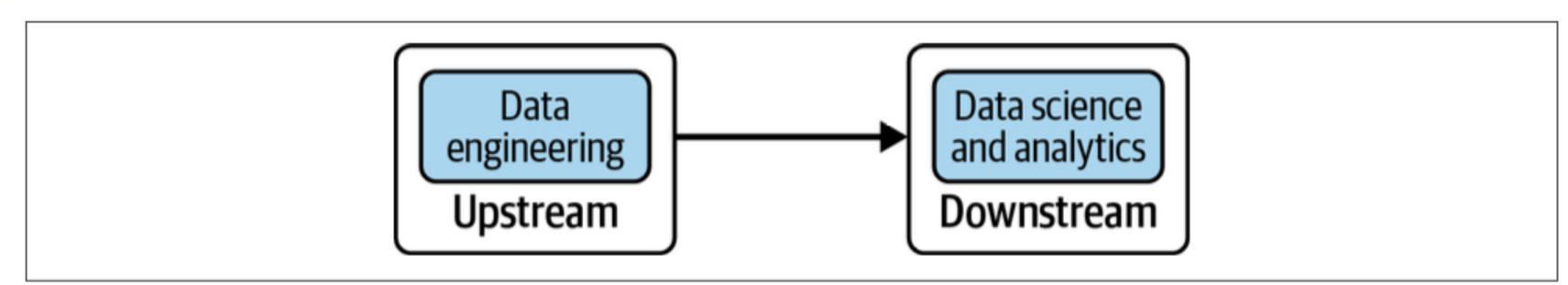
From OLTP to OLAP

Rahul Dave(@rahuldave), Univ.Ai

Purpose of Data Engineering



- The goal is to help the managers of an enterprise make decisions: by visualizing transaction data, incorporating external data, preparing data for analytics and predictive models, storing machine learning features, etc.

- The second goal is to quantify the effect of these decisions. This may be done by comparing predictions with reality, running A/B tests, collecting data from sensors and customers, instrumenting marketing campaigns, etc

Data Engineering Lifecycle

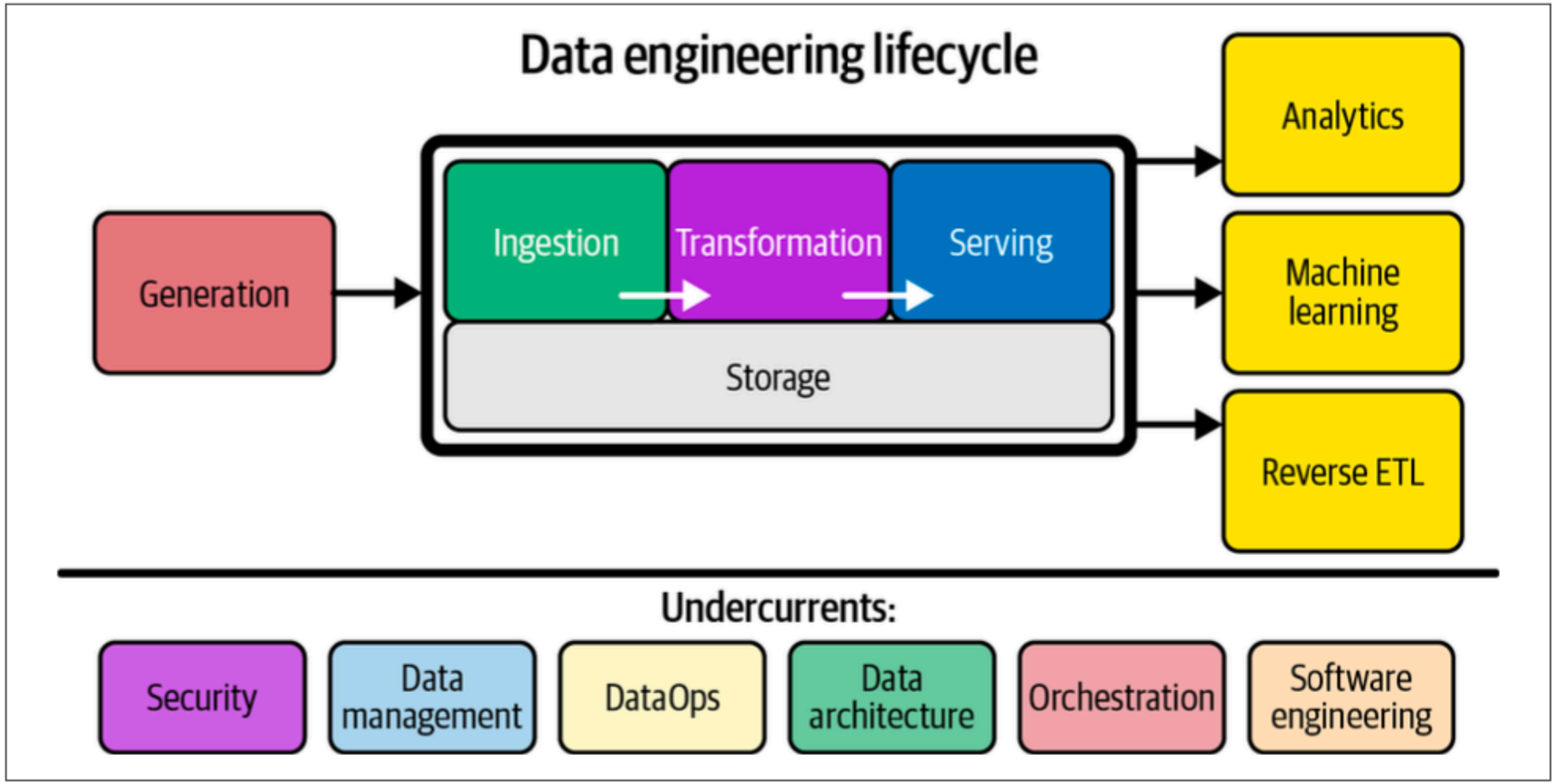
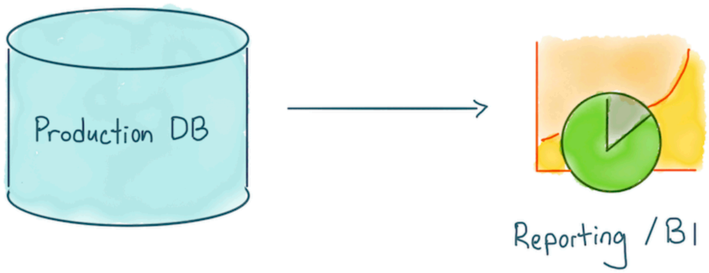
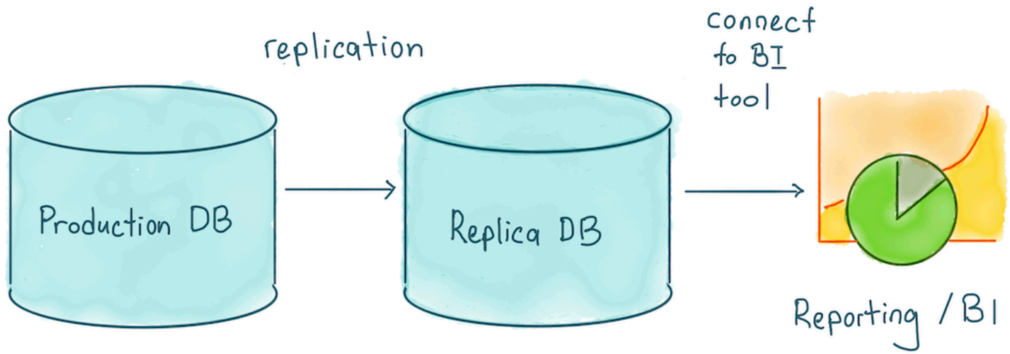


Figure 1-1. The data engineering lifecycle

Organizational Evolution

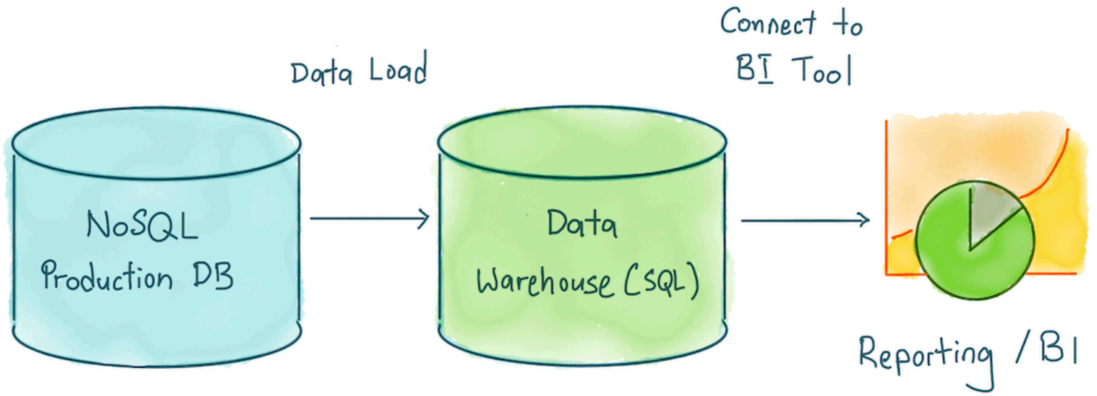


Start by using the production database as your analytics database



Move to using a replica of the production database as your analytics database

Multiple production databases; use a warehouse instead and not affect their operation and to enable efficient analytics



Univ.AI

Four Reasons To Get A Data Warehouse

- Reason 1: You need to analyse data from different sources
- Reason 2: You need to separate analytical from transactional data.
- Reason 3: Your original data source isn't suitable for querying (NoSQL, for instance!)
- Reason 4: To improve the performance of your most-used queries (think: transformed summary tables!)

Created by Holistics 

holistics.io

Evolution of the Data Warehouse

- Started very resource constrained in the 80s. RDBMS. Low memory and low storage. (DB2/Oracle)
- Thus people used OLAP cubes instead, which were de-normalized array-of-array structures (Cognos)
- Resource constraints eased, but data continued to grow. People used RDBMS (Postgres) for OLAP.
- In the mid to late 2000s, columnar databases were invented which made OLAP workloads much faster. (Vertica, MonetDB)
- Data rates were growing exponentially, so people also moved to co-locating compute with data on Hadoop like systems. (Hadoop/HDFS)
- The cloud happened. The ease of autoscaling up both storage and compute resources led to the separation of storage and compute. Large warehouses were implemented on object storage in the cloud. (Redshift, BigQuery, Snowflake)

Transactional DBs vs. Analytics DBs

Data:

- Many single-row writes
- Current, single data

Queries:

- Generated by user activities; 10 to 1000 users
- < 1s response time
- Short queries

Data:

- Few large batch imports
- Years of data, many sources

Queries:

- Generated by large reports; 1 to 10 users
- Queries run for hours
- Long, complex queries

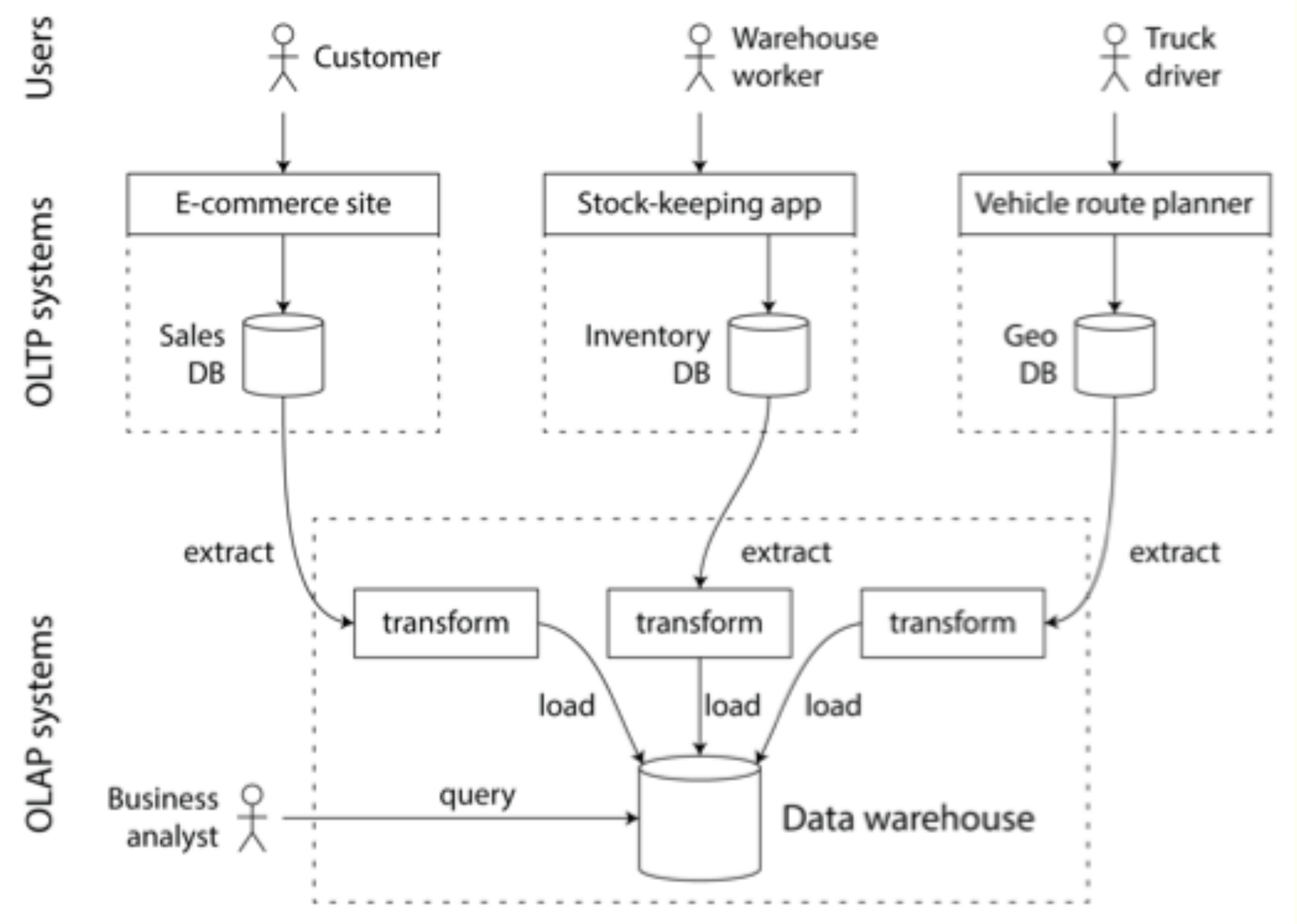
Ref: http://www.slideshare.net/PGExperts/really-big-elephants-postgresql-dw-15833438 (slide 5)

Transactional vs Analytics workloads

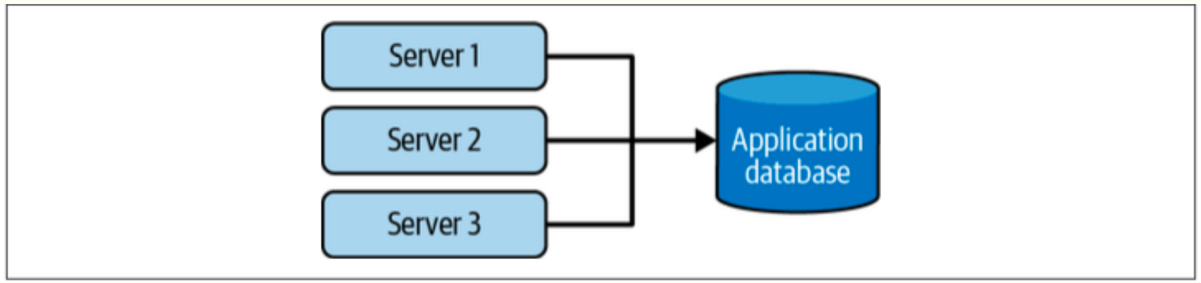
- queries retrieve single record or small number of records
- queries take a very short time
- concurrent queries depend upon number of concurrent readers like web site visitors
- lots of small writes
- usually interested in updating or reading a whole row

- typically scanning a large number of rows, typically as a result of some filter
- queries are heavy and take a long time to finish
- concurrent readers are few and limited by the number of analysis pipelines runnng
- usually interested in just a few columns of data

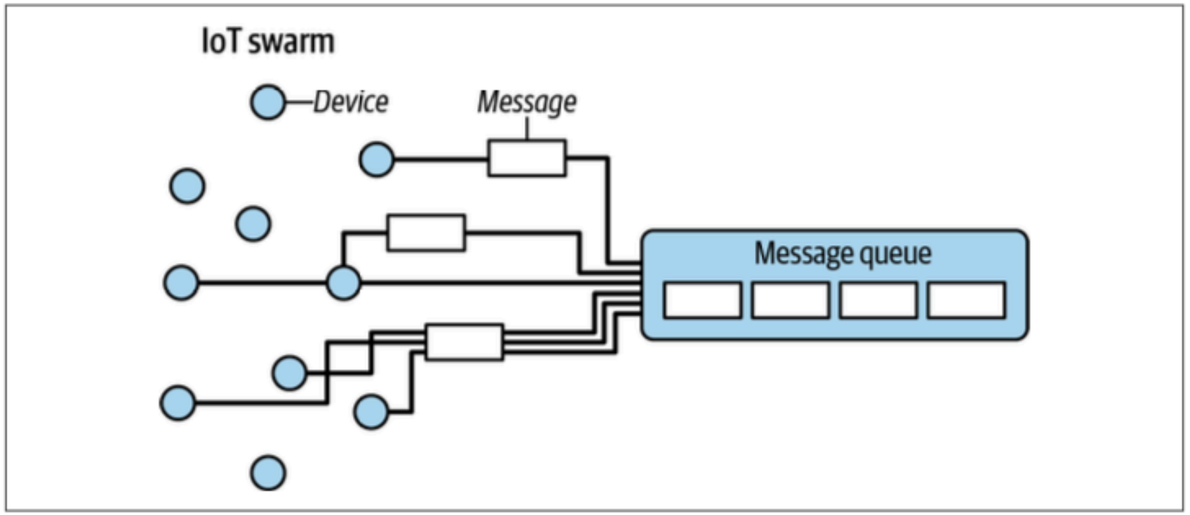
How are the databases used?



- csv, json, txt, etc files
- apis to ERP, CMS, CRM (pipedrive/salesforce/ zendesk), analytics (google, firebase, amplitude), sales, marketing (facebook ads, google ads, dblclick) and other SAAS systems in REST/ graphql. grpc.
- an OLTP database for sales and other transactions
- website log files
- OLAP systems fed by a reverse ETL
- stream messages from IOT systems or webhooks via event streaming systems and message queues
- noSQL databases such as kv, document, graph, time-series and search

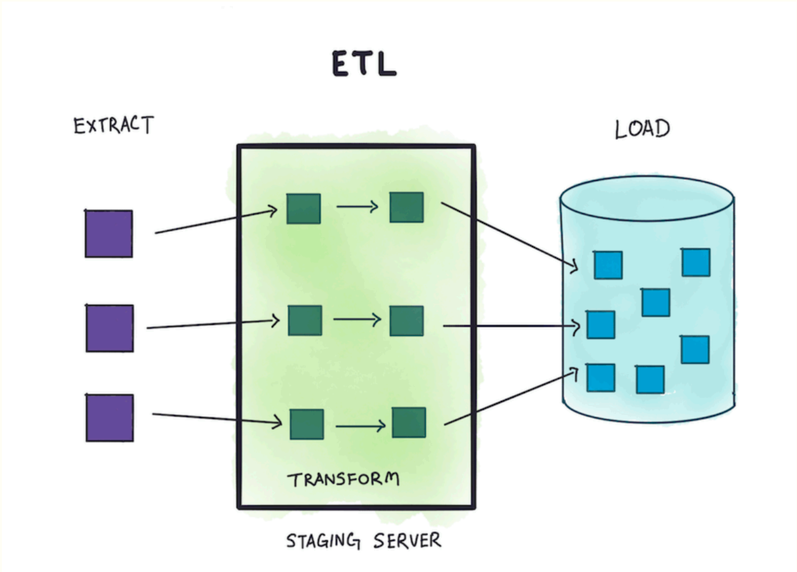


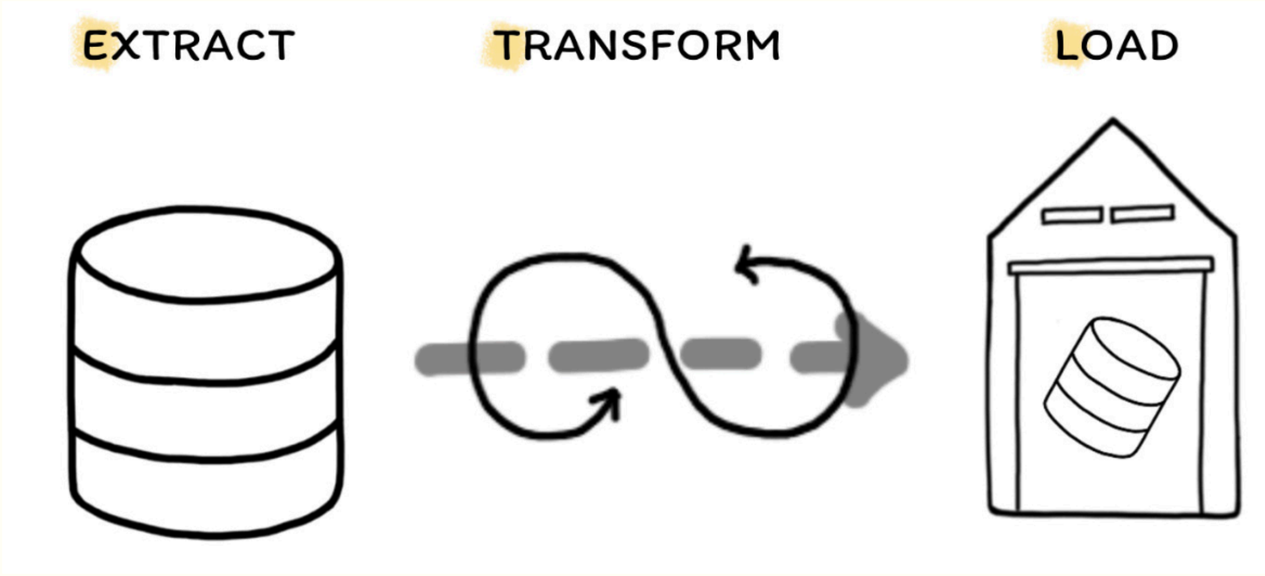
Sources



ETL

- extract data from sources via API, SQL, or CDC
- extracted data is transformed in the memory of the ETL tool
- transformed data is then loaded in the warehouse/ data store.
- often a dedicated staging server is used where the ETL process is run.
- the ETL process may even be run on a cluster using Apache Spark





ETL from multiple sources

- you will need to write ETL scripts from your own databases
- then you will need to write API clients for mutiple endpoints, worry about their auth, and all of this.
- dont do it. it will suck your soul dry. try and outsource this.
- open source: meltano
- commercial: Alooma, HevoData, StitchData, Talend, Pentaho

Olap Cubes

- one of the biggest shifts of the last decade is the move away from specifically engineered olap cubes to more general
- Why does a standard rdbms have problems? Consider the query to the right (N unions for aggregating over N dimensions), or the creation of a cross-tabulation, where the groupings and thus unions scales as 2^dim (here dim=2).
- OLAP cubes were a solution to this problem in the early days of low memory and storage: create a N-dimensional nested array.
- The catch was that the cube thus created depended upon the question asked. And had to be re-made if we wanted a different aggregate

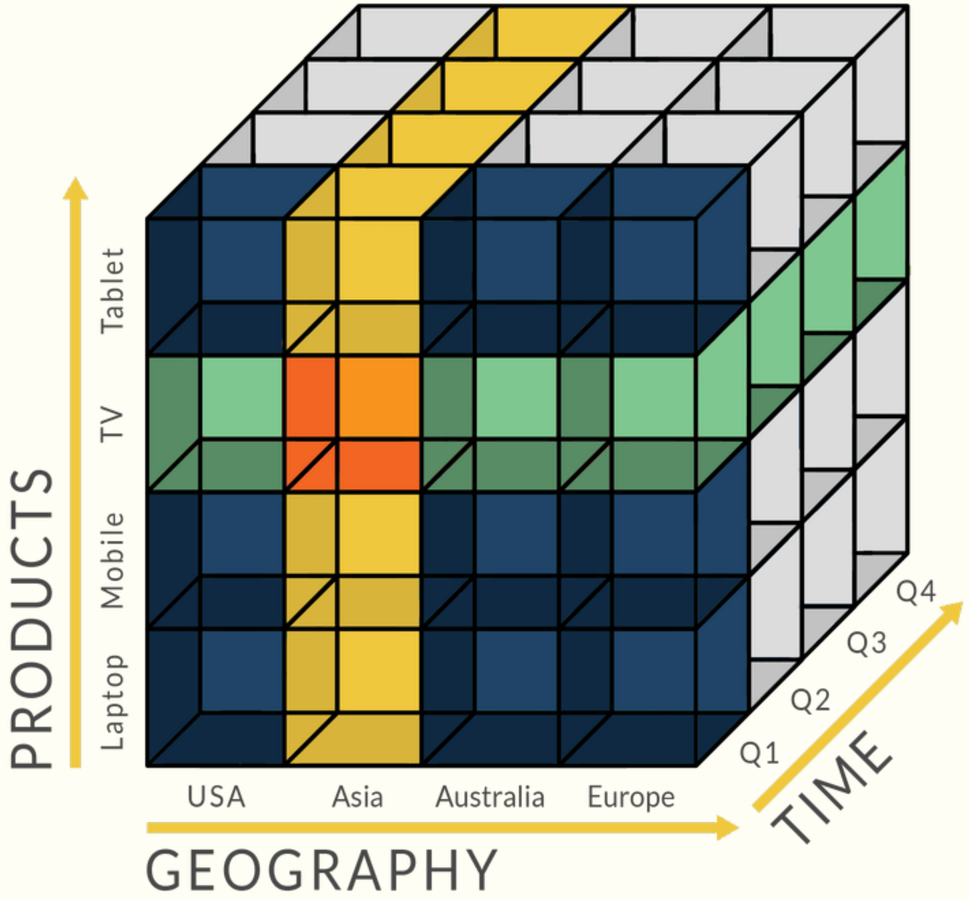
```sql
SELECT Model, ALL, ALL, SUM(Sales)
FROM Sales
WHERE Model = 'Civic'
GROUP BY Model
UNION
SELECT Model, Month, ALL, SUM(Sales)
FROM Sales
WHERE Model = 'Civic'
GROUP BY Model, Year
UNION
SELECT Model, Year, Salesperson, SUM(Sales)
FROM Sales
WHERE Model = 'Civic'
GROUP BY Model, Year, Salesperson;
```

Table 5: Chevy Sales Cross Tab

Chevy	1994	1995	total (ALL)
black	50	85	135
white	40	115	155
total (ALL)	90	200	290

OLAP Cubes

- For huge amounts of data even the OLAP cube must be persisted to disk
- Data engineers would run ETL to create these cubes and then hand them to data analysts
- What changed? (1) Cheap memory and Disk. (2) Massively parallel processing: spin up multiple cpus on a cloud to do queries (3) Columnar databases

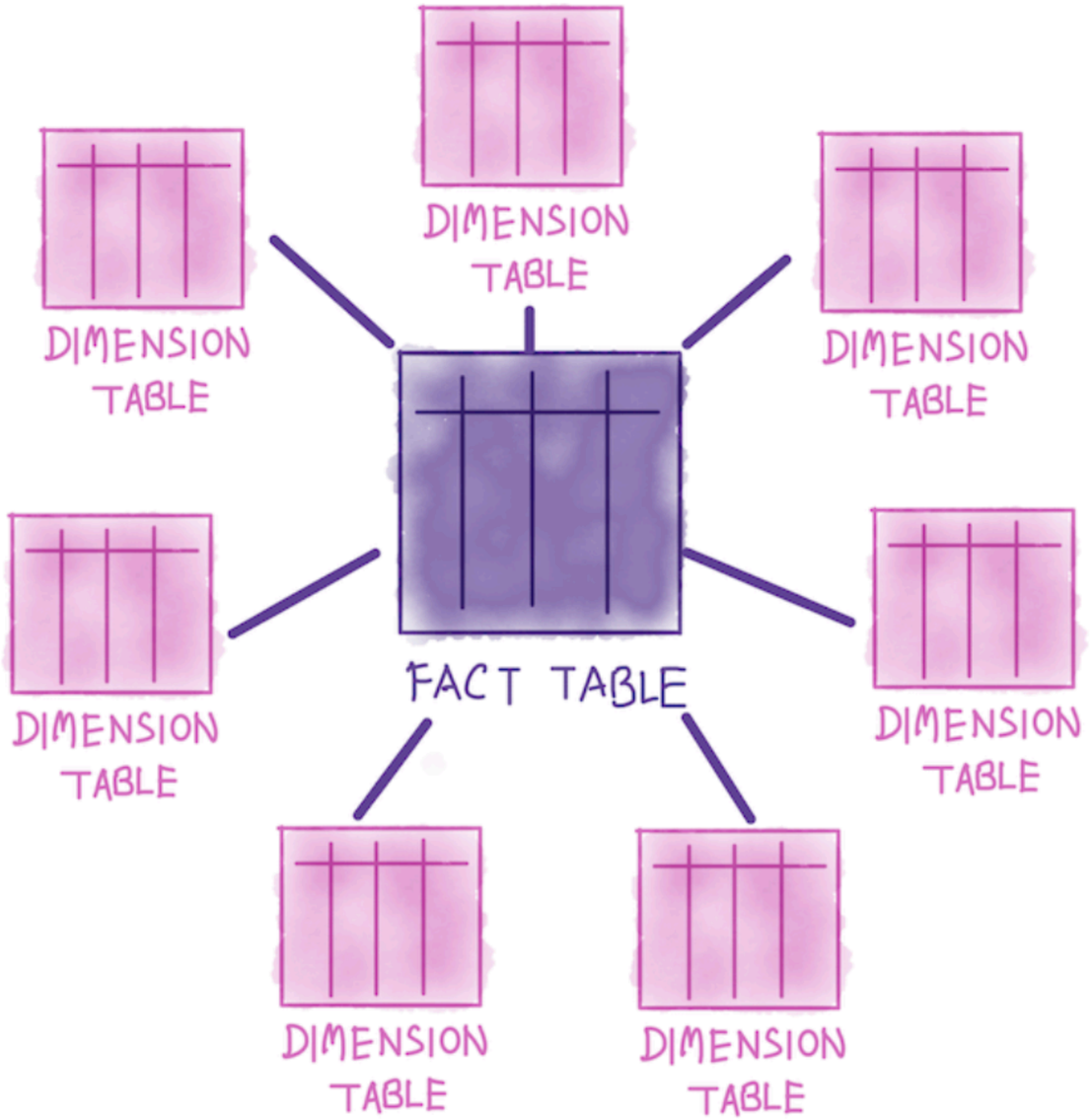


Warehouse Schema

- Before the data warehouse, analytics would often occur in the source database itself, bogging down the system
- There are different approaches but the most famous ones are Kimball (1996) and Immon (1990) and Data Vault (early 2000s)
- Immon's is highly normalized, Kimball's in relatively denormalized
- Modern systems use a combination of Kimball and wide denormalized tables.

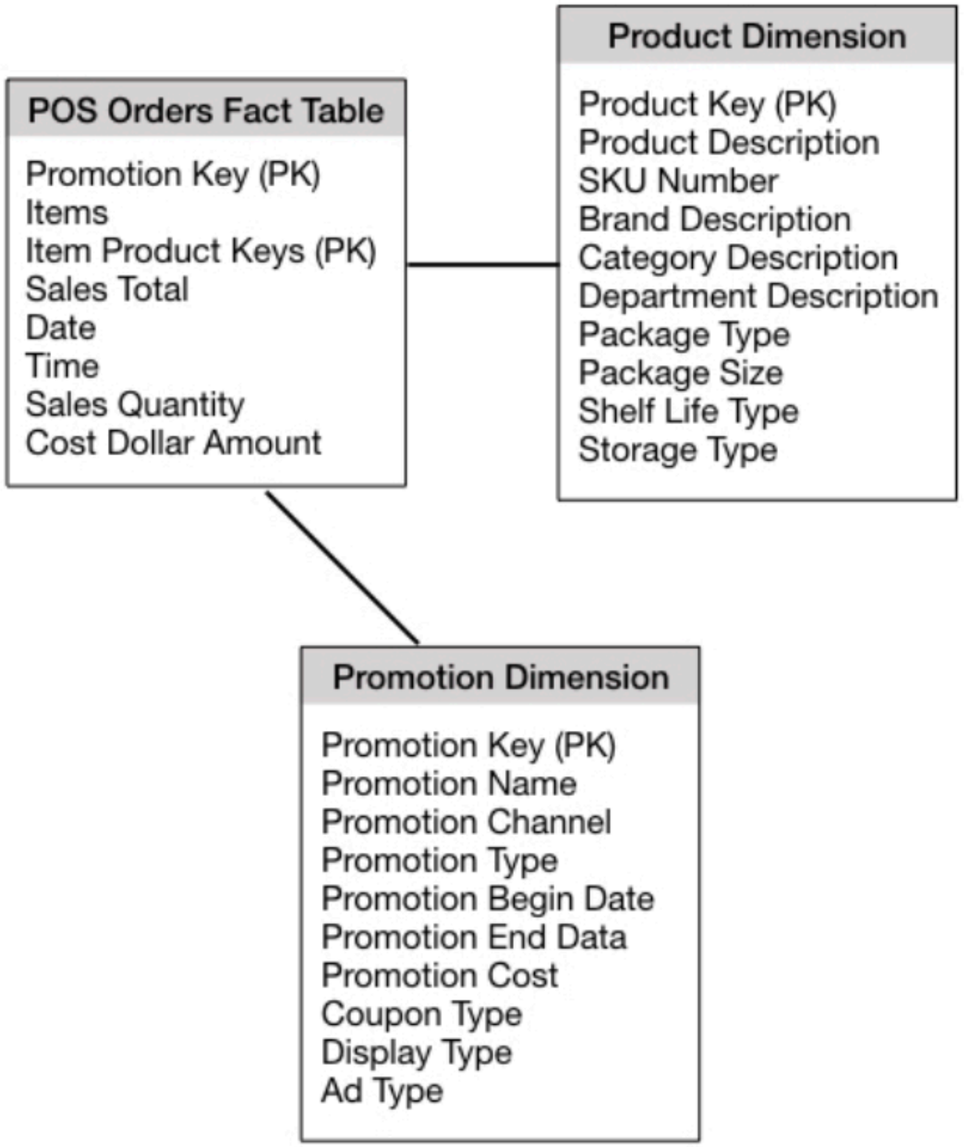
From Normalization to De-Normalization

- Normalization is important in OLTP databases for reducing the surface area of transactions: less the surface area, less the locking, less the rollback.
- But pointers are not useful in datasets you want to feed to BI software such as Tableau or machine learning software such as sklearn
- There you sometimes want to de-normalize datasets by joining multiple tables
- Usually you will want to do an inner join. However outer joins are important when you need your full data to be represented, even if they lead to more nulls.
- Think of your problem and ask: what join do I need?



Kimball Dimensional Modeling

- This is a Star Schema. We have a **fact table** surrounded by **dimension table**s.
- The focus is less on normalization. Indeed dimension tables can be quite denormalized
- Facts are the primary metrics of a business process. These metrics have attributes, which are the dimensions
- The start schema is flexible, extensible, and performant in row-oriented RDBMS (star join).



Facts and Dimensions

- Size of fact table grows rapidly
- Size of product dimension only grows when a new SKU is added.
- The promotion dimension grows even more slowly

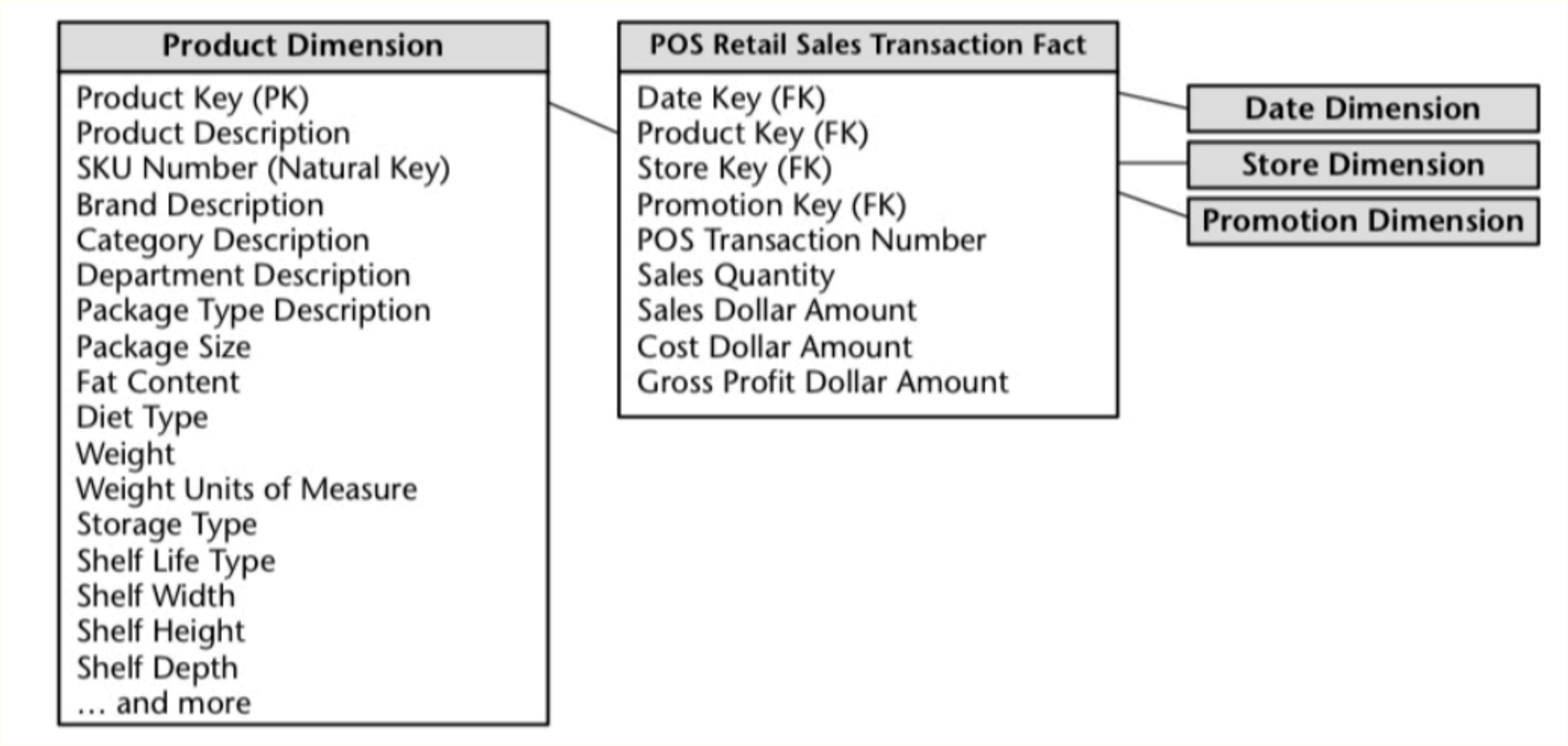
Kimball Modeling

1. Pick a business process to model (verbs).

2. Decide on the grain. The level of grain chosen should be the most atomic possible; we can always coarse grain it later. For e.g., in the POS example before, we should model at the line-items in every order. Then you can answer: which products sold best in the day at the store?

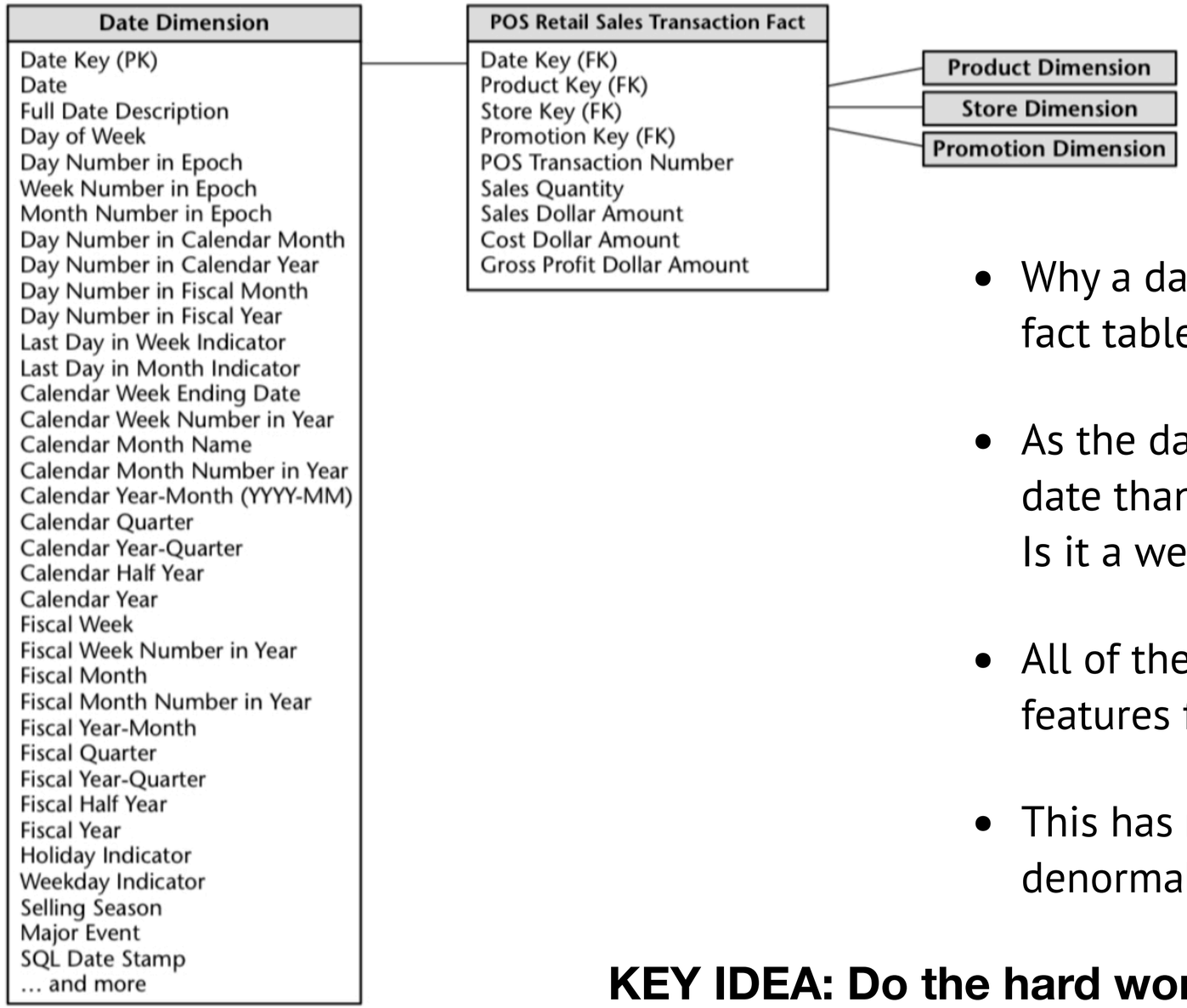
3. Choose the dimensions that apply to each fact row. Ask: how does a business person describe the attributes of each fact, not just for characterization, but also for querying

4. What numeric measures ought to populate each fact row? For example, you might want to figure the profit per item..



Notice how numerical items are carefully chosen in the fact table

Stuffing features into a dimension



- Why a date dimension? wouldnt a simple date in the fact table do?
- As the date dimension shows, there is much more to a date than just the date: how close are we to Christmas? Is it a weekend? Is it in the first quarter? Is it a holiday?
- All of these make for great queries, and even great features for a machine learning system
- This has repetition and is in this sense quite denormalized!

KEY IDEA: Do the hard work now, to make it easy to query later!

More on Kimball

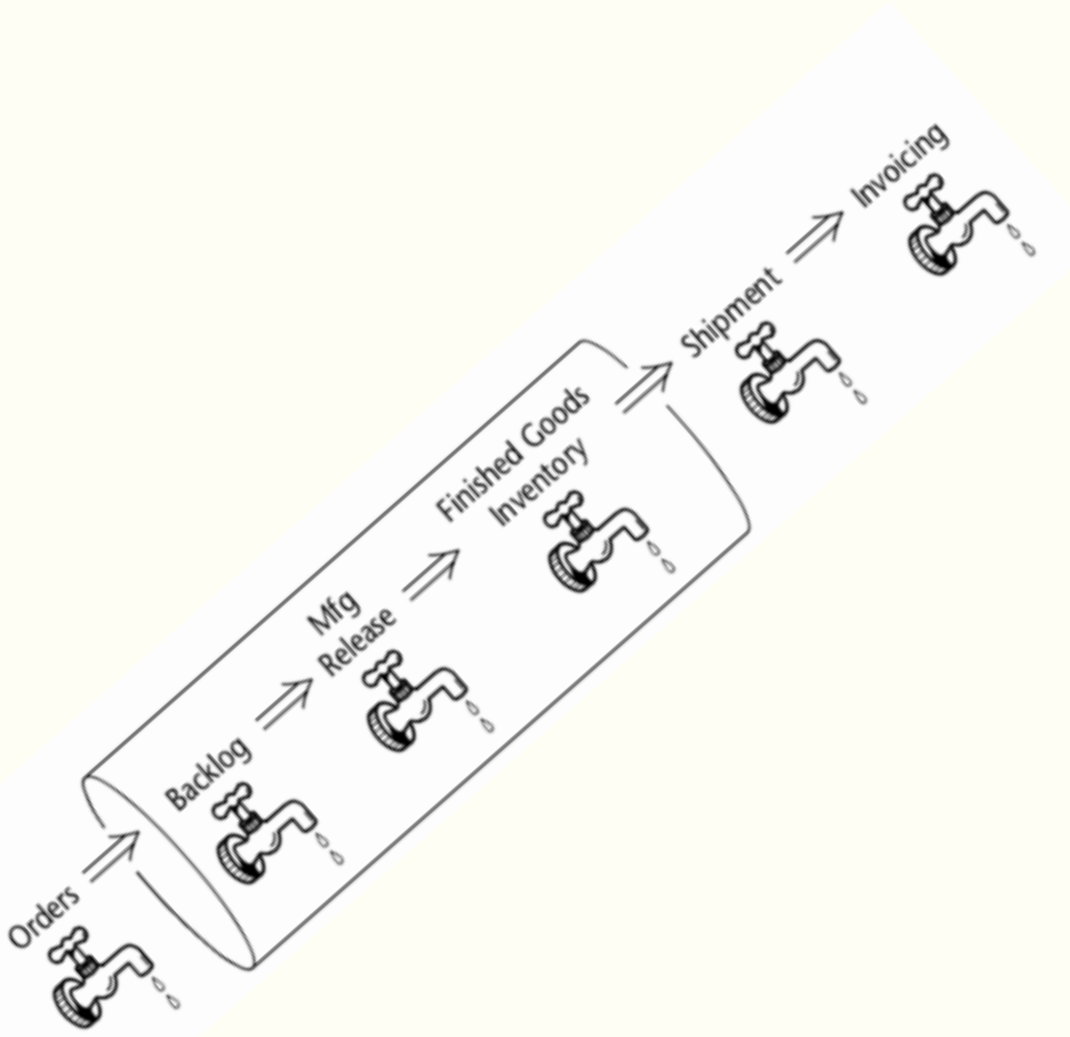
- star schemas do not reflect a particular report: you want to make them general enough to answer multiple questions
- make your dimension tables reusable in multiple star schema to avoid drifting business definitions and implement data integrity. Such dimensions are called conformed dimensions
- besides the basic transactional fact table, you will want to support aggregate fact tables. Our original fact tables are at the smallest grain that makes sense.

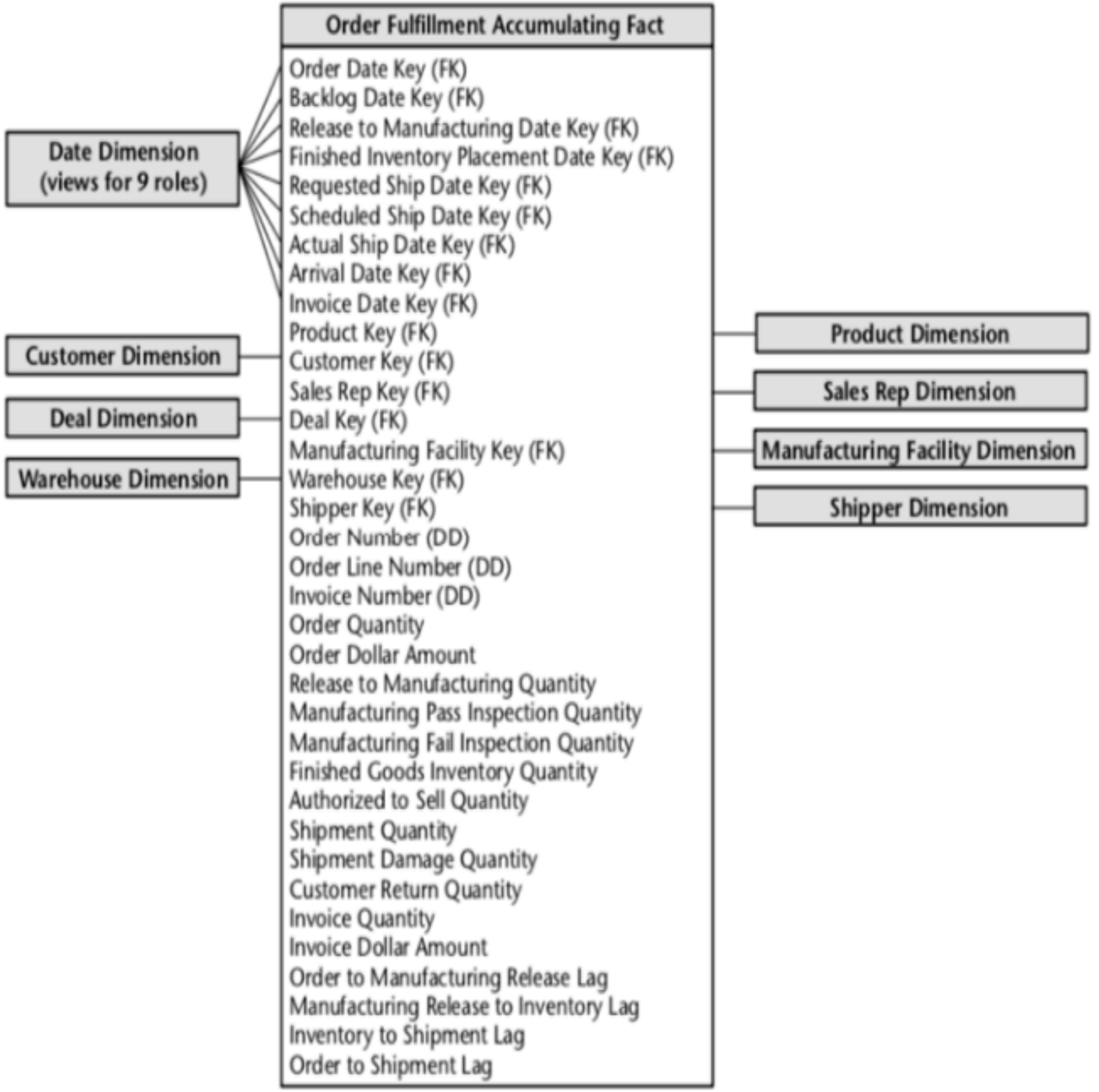
Aggregated Fact Tables

- **Periodic snapshot fact** tables have the grain as the period, not the individual transaction. For example, you might aggregate all sales over a day, or a monthly inventory tally. If no transactions occur for a fact, insert a null.

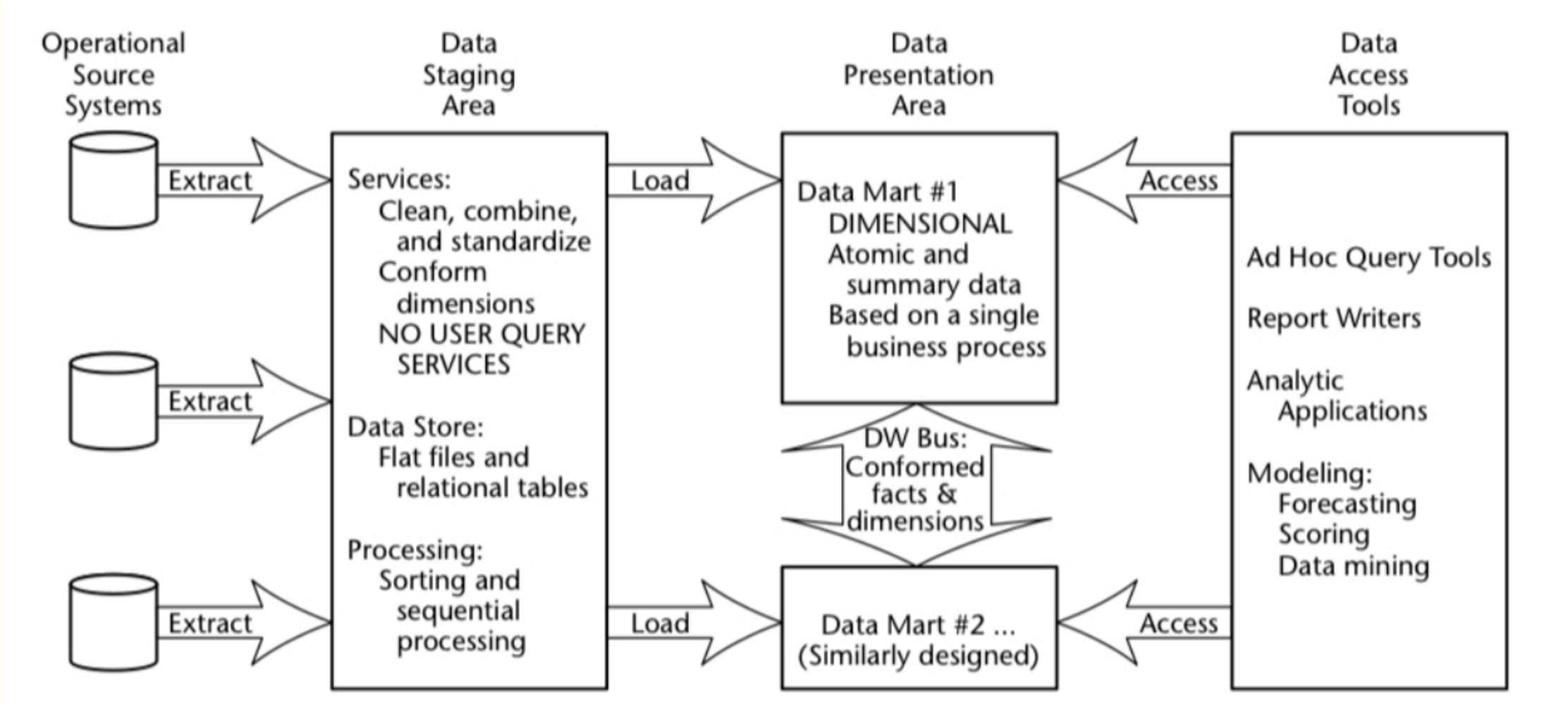
- These tables can combine aggregates from multiple fact tables

- **Accumulating Snapshot Tables** measure velocity within a business process. The lifetime of an order, for example. They are often filled in multiple times.

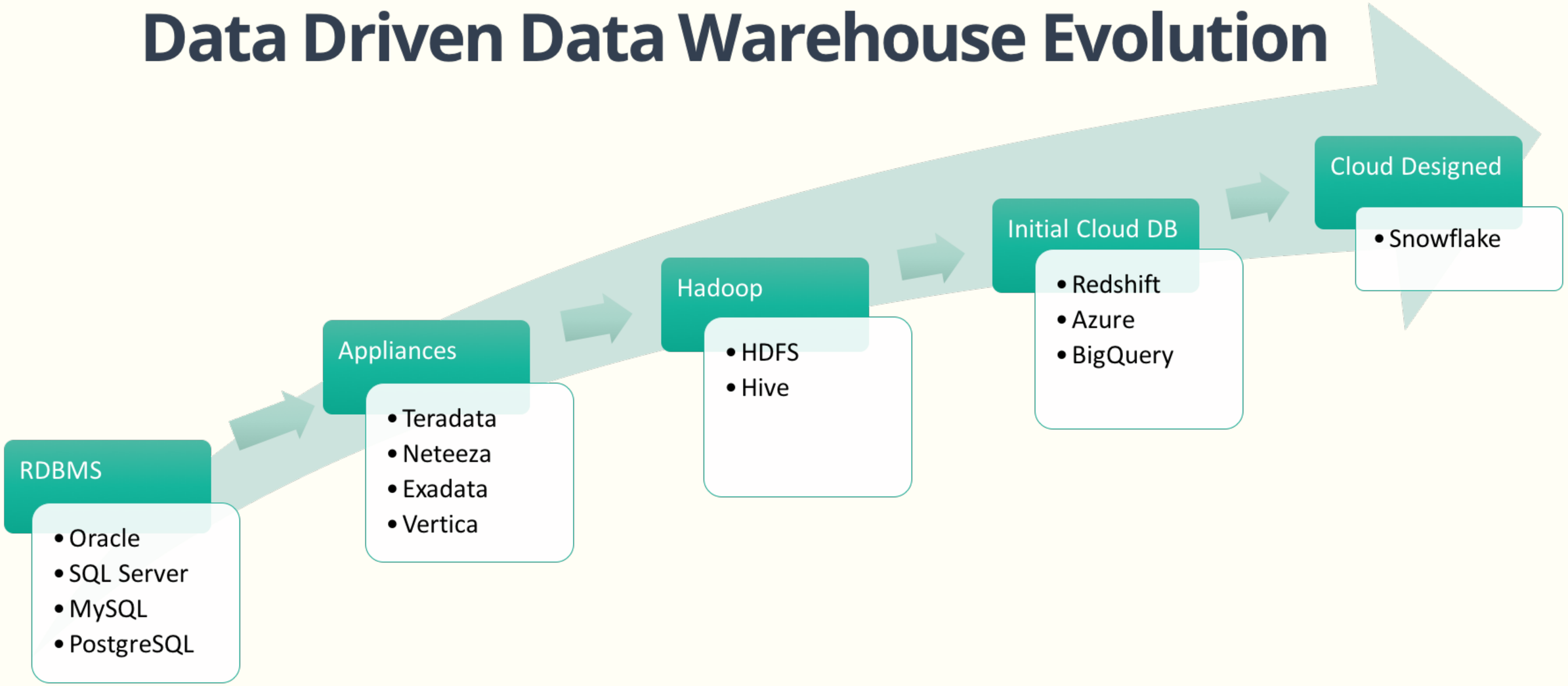

Orders
Backlog
Mfg Release
Finished Goods Inventory
Shipment
Invoicing


Order Fulfillment Accumulating Fact
Order Date Key (FK)
Backlog Date Key (FK)
Release to Manufacturing Date Key (FK)
Finished Inventory Placement Date Key (FK)
Requested Ship Date Key (FK)
Scheduled Ship Date Key (FK)
Actual Ship Date Key (FK)
Arrival Date Key (FK)
Invoice Date Key (FK)
Product Key (FK)
Customer Key (FK)
Sales Rep Key (FK)
Deal Key (FK)
Manufacturing Facility Key (FK)
Warehouse Key (FK)
Shipper Key (FK)
Order Number (DD)
Order Line Number (DD)
Invoice Number (DD)
Order Quantity
Order Dollar Amount
Release to Manufacturing Quantity
Manufacturing Pass Inspection Quantity
Manufacturing Fail Inspection Quantity
Finished Goods Inventory Quantity
Authorized to Sell Quantity
Shipment Quantity
Shipment Damage Quantity
Customer Return Quantity
Invoice Quantity
Invoice Dollar Amount
Order to Manufacturing Release Lag
Manufacturing Release to Inventory Lag
Inventory to Shipment Lag
Order to Shipment Lag
Date Dimension (views for 9 roles)
Customer Dimension
Deal Dimension
Warehouse Dimension
Product Dimension
Sales Rep Dimension
Manufacturing Facility Dimension
Shipper Dimension

Back in the day...



Data Driven Data Warehouse Evolution



..meanwhile the world changes...

The two phases

- Phase 1: the first columnar databases arive: C-store (2005) which gets commercialized as Vertica, and open-source MonetDB. These are expensive, need big honking machines, and run on-prem. Rich People can afford them.

- The interim, that sees the rise of Hadoop, and the co-location of compute with storage. ETL scripts now used CDC or JDBC against a replica database, and do computations, and later SQL with Hive and Presto (2013ish) on databases and data stored in formats like Avro, Parquet, and ORC on distributed storage such as HDFS

- Phase2: everything moves to the cloud, and the spinning up of elastic virtual machines and containers on EC2 and others ushers in the Massively Parallel Processing Era which allows scaling storage separately from compute: we get distributed SQL. SQL now operates on warehouses AND files in the cloud.

Columnar Databases

- Store each column separately
- Have a higher read efficiency as only a few columns of contiguous or run-encoded data need to be read
- compress better especially if the cardinality of the columns is not high thus allowing more data to be loaded into memory
- columnar data have higher sorting and indexing efficiency ans may even admit multiple sort orders

Row vs Column

OrderId	CustomerId	ShippingCountry	OrderTotal
1	1258	US	55.25
2	5698	AUS	125.36
3	2265	US	776.95
4	8954	CA	32.16

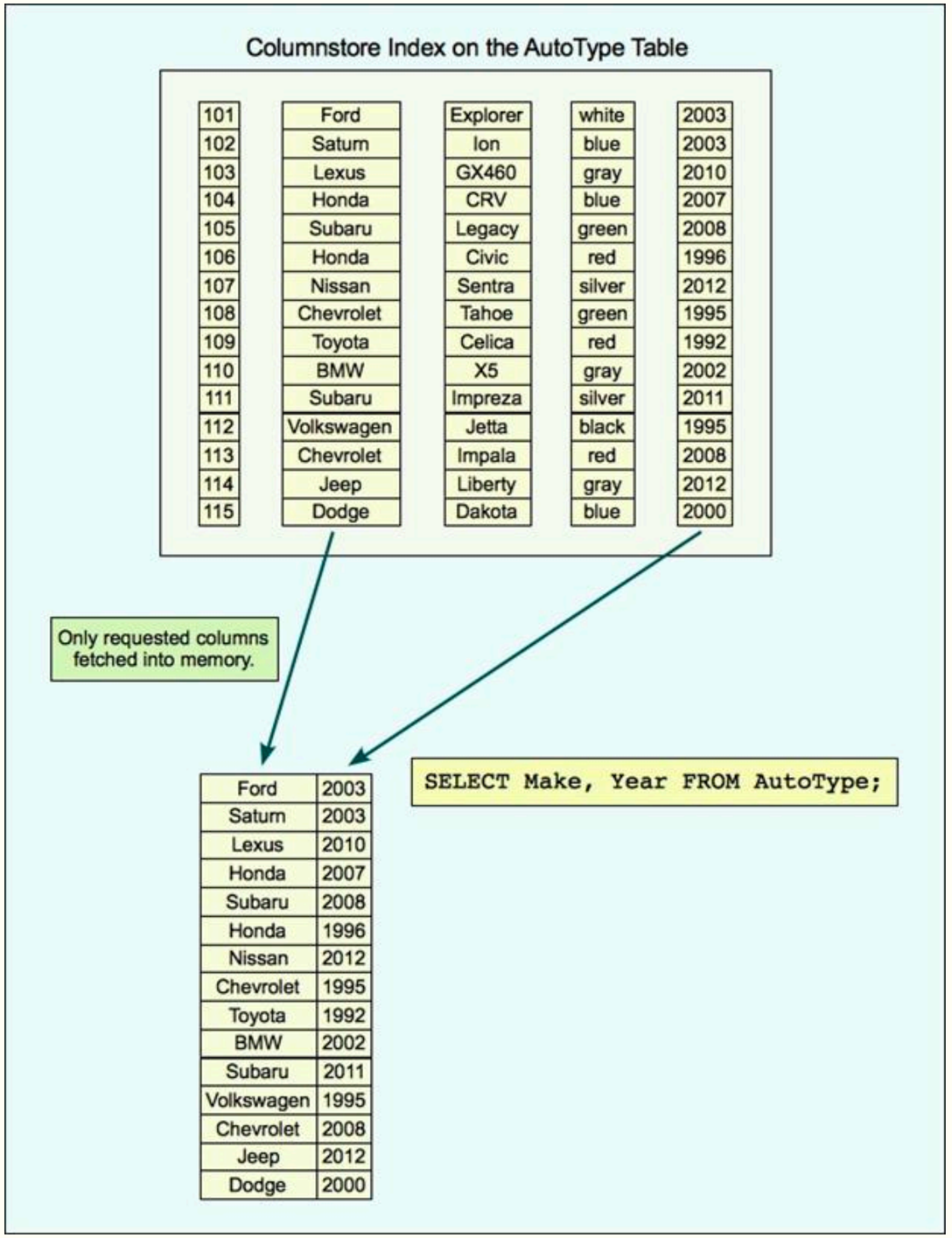
Block 1	1, 1258, US, 55.25
Block 2	2, 5698, AUS, 125.36
Block 3	3, 2265, US, 776.95
Block 4	4, 8954, CA, 32.16

OrderId	CustomerId	Shipping Country	Order Total	Customer Active
1	1258	US	55.25	TRUE
2	5698	AUS	125.36	TRUE
3	2265	US	776.95	TRUE
4	8954	CA	32.16	FALSE

Block 1	1, 2, 3, 4
Block 2	1258, 5698, 2265, 8954
Block 3	US, AUS, US, CA
Block 4	55.25, 125.36, 776.95. 32.16
Block 5	TRUE, TRUE, TRUE, FALSE

Column-oriented Storage

- store values from each column together in separate storage
- lends itself to compression with bitmap indexes
- compressed indexes can fit into cache and are usable by iterators
- several different sort orders can be redundantly stored
- writing is harder: updating a row touches many column files
- but you can write an in-memory front sorted store (row or column), and eventually merge onto the disk



Bitmap Indexes

- lends itself to compression with bitmap indexes and run-length encoding. This involves choosing an appropriate sort order. The index then can be the data (great for IN and AND queries): there is no pointers to “elsewhere"
- bitwise AND/OR can be done with vector processing

Column values:

product_sk:	69	69	69	69	74	31	31	31	31	29	30	30	31	31	31	68	69	69

Bitmap for each possible value:

product_sk = 29:	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
product_sk = 30:	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
product_sk = 31:	0	0	0	0	0	1	1	1	1	0	0	0	1	1	1	0	0	0
product_sk = 68:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
product_sk = 69:	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1
product_sk = 74:	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

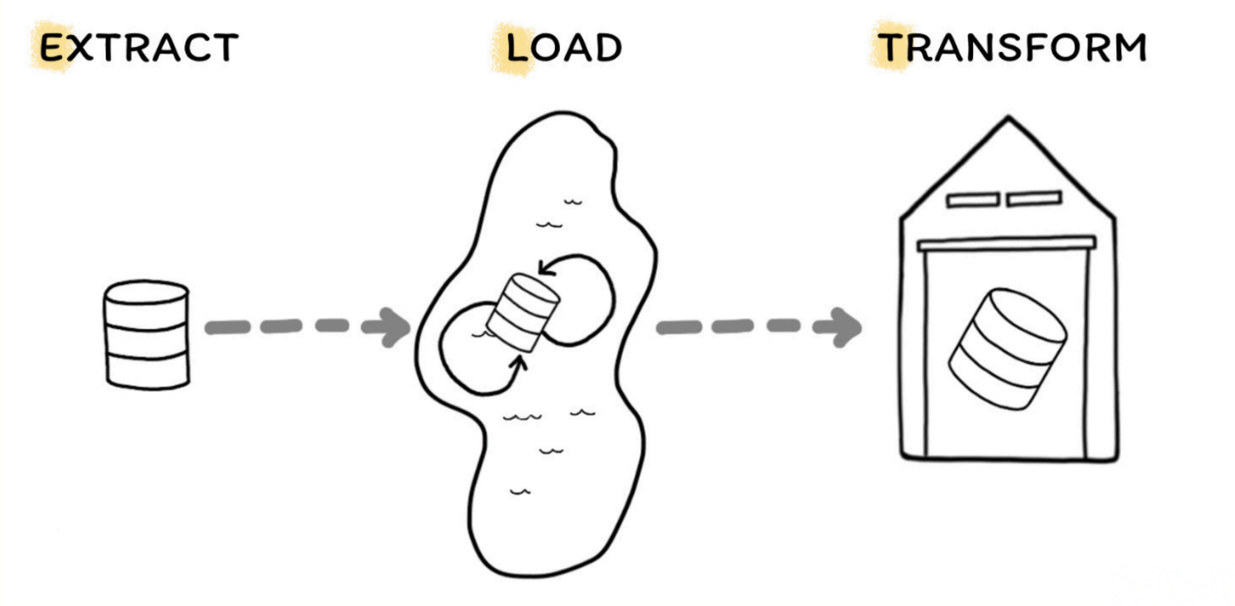
Run-length encoding:

product_sk = 29:	9, 1	(9 zeros, 1 one, rest zeros)
product_sk = 30:	10, 2	(10 zeros, 2 ones, rest zeros)
product_sk = 31:	5, 4, 3, 3	(5 zeros, 4 ones, 3 zeros, 3 ones, rest zeros)
product_sk = 68:	15, 1	(15 zeros, 1 one, rest zeros)
product_sk = 69:	0, 4, 12, 2	(0 zeros, 4 ones, 12 zeros, 2 ones)
product_sk = 74:	4, 1	(4 zeros, 1 one, rest zeros)

Columnar Database choices

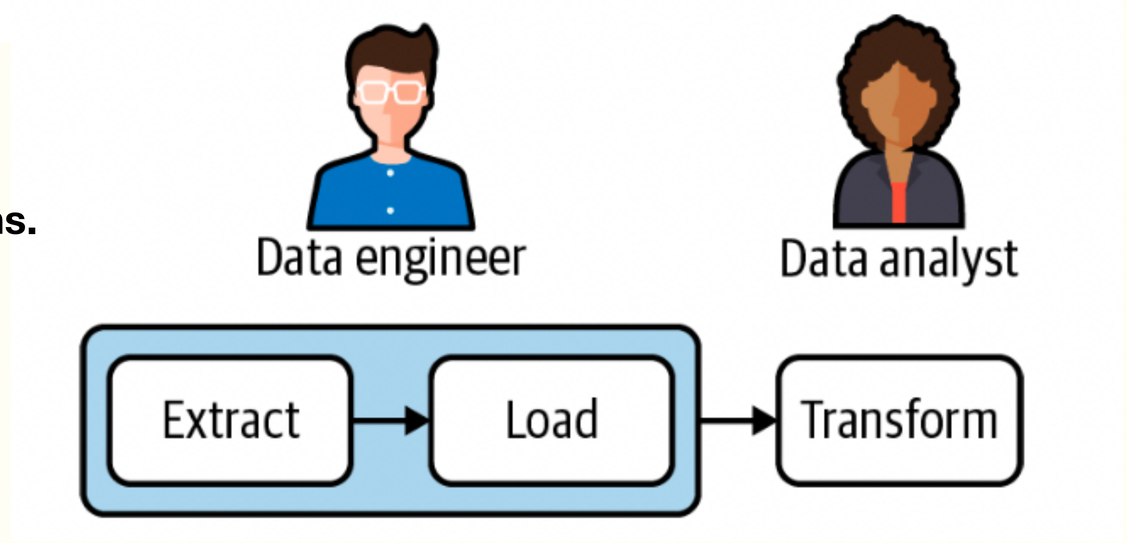
- Cloud: Amazon Redshift: integrated with S3, needs DBA
- Cloud: Google Bigquery: came out of Dremel, auto-resizable, less maintenance
- Cloud: Snowflake: highly scalable, expensive, less maintenance
- Cloud and On-Prem: Yandex Clickhouse (open-source)
- Embedded: Duckdb (open-source)

ETL to ELT

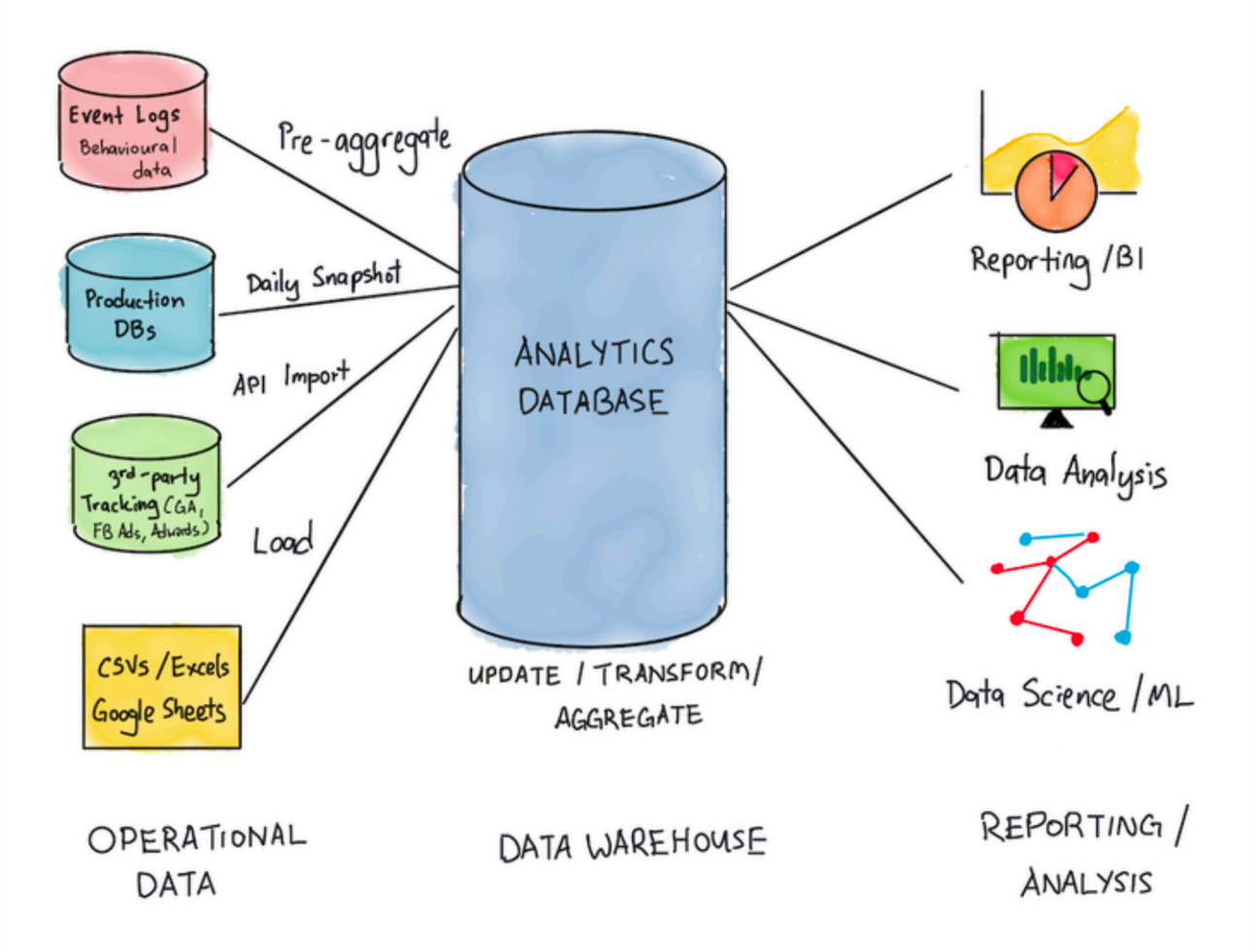


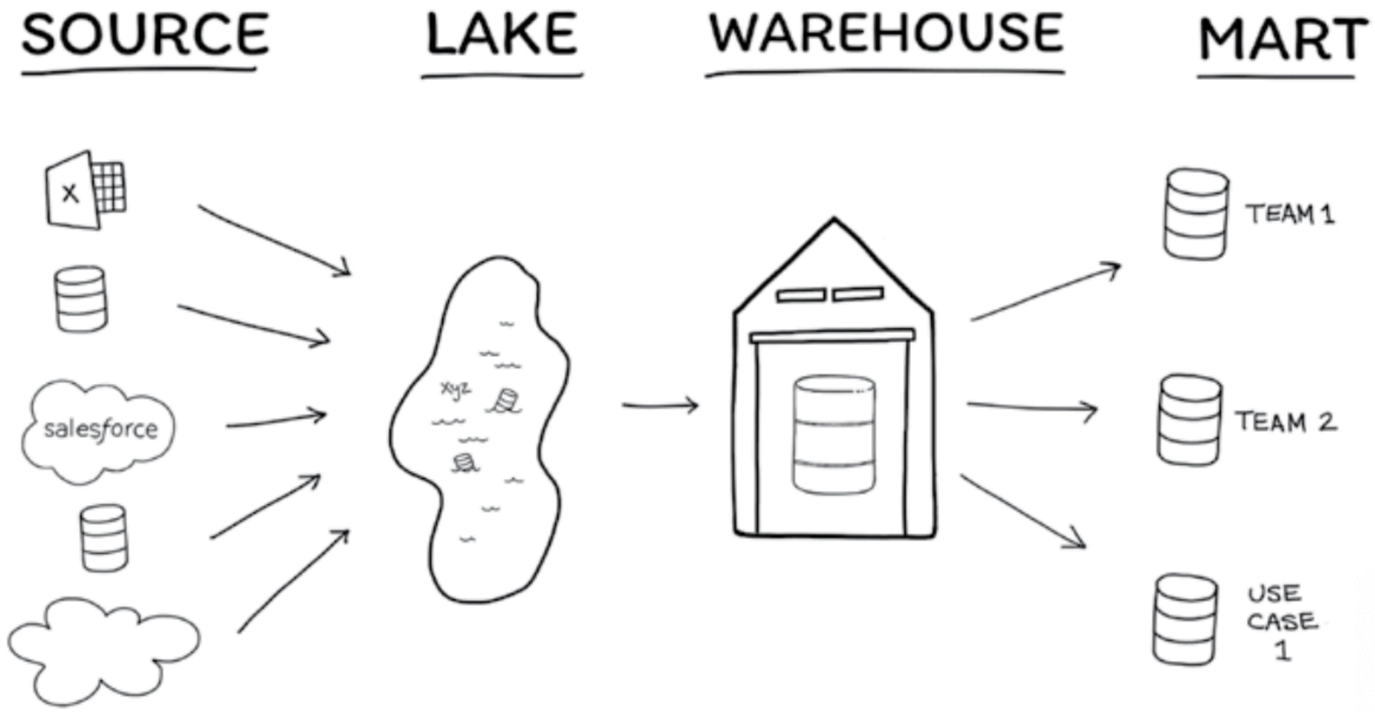
Move stuff into a warehouse or lake first!
This is called a source refreshed table

Now transformations are done IN the warehouse.
The data engineer can do the initial loading and transformations.
Data analysts from client groups (sales, marketing)
can do subsequent transformations



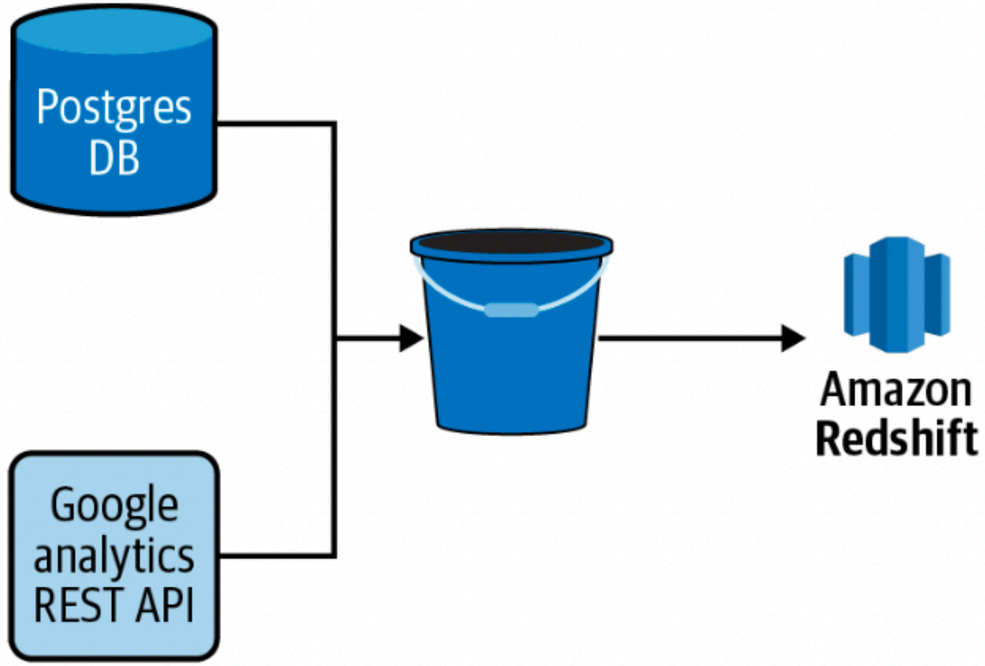
Modern Stack:ELT





A lake is often a useful intermediary. All kinds of data: tabular/image/text can be dumped there. It can also be used for post warehousing data outputs for downstream pursuits such as machine learning.

Analysis using non-SQL code such as using SPARK can also be carried out in the data lake.



More precisely: EtLT

- First just copy the source tables into the warehouse. This is the first *L*. This may even be done in a data lake. These tables are called fully refreshed tables. We then create **Staging Schema**s which are lightly changed versions of source tables from OLTP databases and other sources.

- These would normally be part of *T* in a ETL scenario (with new tables in the warehouse) but are the small *t* in the ELT scenario here. Since these are incomings into a database, we will create lightly changed schema's from OLTP schemas or completely new ones for other sources.

- tables and columns that are unneeded or empty are removed, weird names are improved, and unwanted data is filtered out. NULLS are handled. Words are stemmed. URLs are parsed. Dates and times are rationalized. Data is de-duplicated. This allows for more complex numeric transformations without worrying about these sorts of consistencies.

- This expands the storage needs a lot, but modern warehouses are more than up to it. Storage is cheap.

Extraction

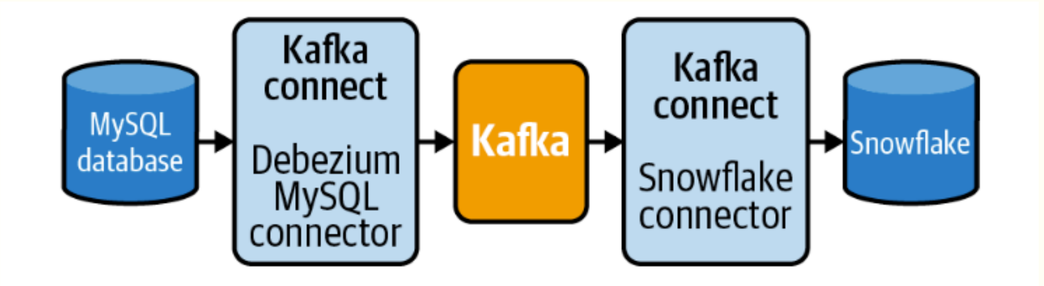
OrderId	OrderStatus	LastUpdated
1	Backordered	2020-06-01 12:00:00
1	Shipped	2020-06-09 12:00:25

- Extraction is carried out on the source database
- The extraction can be a full table extraction which will give you the current snapshot of the extraction
- Or it can be an incremental extraction, which gives you the latest changes on a table. Note that this includes updates to existing rows.
- The extraction is made simple by a last-updated timestamp.
- In the ETL paradigm you would make changes like de-duplication before arrival at lake/warehouse; in ELT you simply copy.

```
CREATE TABLE Orders (
  OrderId int,
  OrderStatus varchar(30),
  LastUpdated timestamp
);

--full extraction
SELECT * FROM Orders;

-- incremental extraction

last_extraction_run = SELECT MAX(LastUpdated) FROM warehouse.Orders;

SELECT *
FROM Orders
WHERE LastUpdated > {{ last_extraction_run} };
```

CDC Extraction

- You can also extract data via **Change Data Capture** (CDC). This uses the source database's WAL or binlog to capture all events.
- The WAL is typically used to replicate database tables in a cluster without having to resort to a SQL query. Here we can hook into the replication system and copy events.
- As a result we will get ALL the events, including deletes, and must remove these when reconstituting the source database in the warehouse.



EventType	OrderId	OrderStatus	LastUpdated
insert	1	Backordered	2020-06-01 12:00:00
update	1	Shipped	2020-06-09 12:00:25
delete	1	Shipped	2020-06-10 9:05:12

Loading

```
-- this example is for Redshift with CDC

CREATE TABLE public.Orders (
  OrderId int,
  OrderStatus varchar(30),
  LastUpdated timestamp,
  EventType varchar(30)
);
-- full load from csv/parquet
TRUNCATE public.Orders;
COPY public.Orders
FROM 's3://bucket-name/full-dump-2022-10-01.csv'
iam_role 'arn:aws:iam::222:role/RedshiftLoadRole';

-- incremental load from csv/parquet
COPY public.Orders
FROM 's3://bucket-name/incremental-dump-2022-09-01_to_2022-10-01.csv'
iam_role 'arn:aws:iam::222:role/RedshiftLoadRole';
```

- Loading can be carried out from a file in the lake (shown here), or directly in an ETL process (using JDBC drivers or SQLAlchemy at either end).
- Loading from CDC can be directly done via Kafka or via a file written out by the CDC process (example shown here).

Transformations in the warehouse

```sql
-- transformation from fully refreshed models in the warehouse

-- warehouse tables

CREATE TABLE Orders ( -- order fact
  OrderId int,
  OrderStatus varchar(30),
  OrderDate timestamp,
  CustomerId int,
  OrderTotal numeric
);

CREATE TABLE Customers ( -- customer dimension
      CustomerId int,
      CustomerName varchar(20),
      CustomerCountry varchar(10)
);
```

```sql
-- metrics needed: revenue/country/month, orders/day
-- fully refreshed tables example
CREATE TABLE IF NOT EXISTS order_summary_daily (
  order_date date,
  order_country varchar(10),
  total_revenue numeric,
  order_count int
);
INSERT INTO order_summary_daily
    (order_date, order_country, total_revenue, order_count)
SELECT
    o.OrderDate AS order_date,
    c.CustomerCountry AS order_country,
    SUM(o.OrderTotal) as total_revenue,
    COUNT(o.OrderId) AS order_count
FROM Orders o
INNER JOIN Customers c on
    c.CustomerId = o.CustomerId
GROUP BY o.OrderDate, c.CustomerCountry;
```

Final Metrics

```sql
-- metrics needed: revenue/country/month, orders/day
-- fully refreshed tables example

-- How much revenue was generated from orders
-- placed from a given country in a given month?
SELECT
    DATE_PART('month', order_date) as order_month,
    order_country,
    SUM(total_revenue) as order_revenue
FROM order_summary_daily
GROUP BY
      DATE_PART('month', order_date),
      order_country
ORDER BY
      DATE_PART('month', order_date),
      order_country;
```

```sql
-- metrics needed: revenue/country/month, orders/day
-- fully refreshed tables example

-- How many orders were placed on a given day?
SELECT
    order_date,
    SUM(order_count) as total_orders
FROM order_summary_daily
GROUP BY order_date
ORDER BY order_date;
```