Putting it all together

Systems design
1. Quick recap of scale

2. A full systems example putting things together
   ▶ How to compute data sizes?
   ▶ Computing a large join?
   ▶ Building an index
**Data models**

**Structured**
(e.g. ads, purchases, product tables)
[aka relational tables]

```
<table>
<thead>
<tr>
<th>ID</th>
<th>user_id</th>
<th>profession</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>&quot;Developer&quot;</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>&quot;Engineer&quot;</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>ID</th>
<th>user_id</th>
<th>name</th>
<th>version</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>1</td>
<td>&quot;MyApp&quot;</td>
<td>1.0.4</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>&quot;DocFinder&quot;</td>
<td>2.5.7</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>ID</th>
<th>user_id</th>
<th>make</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>1</td>
<td>&quot;Bentley&quot;</td>
<td>1973</td>
</tr>
<tr>
<td>31</td>
<td>1</td>
<td>&quot;Rolls Royce&quot;</td>
<td>1965</td>
</tr>
</tbody>
</table>
```

**Semi-structured**
(e.g. user profile, web site activity)
[aka JSON documents]

```
last_name: "Jones",
cell: "516-555-2048",
city: "Long Island",
year_of_birth: 1986,
location: {
  type: "Point",
  coordinates: [-73.9876, 40.7574]
},
profession: ["Developer", "Engineer"],
apps: [
  {
    name: "MyApp",
    version: 1.0.4,
  },
  {
    name: "DocFinder",
    version: 2.5.7
  }
],
cars: [
  {
    make: "Bentley",
    year: 1973,
  },
  {
    make: "Rolls Royce",
    year: 1965
  }
]```
In reality, a mix of systems -- e.g., Amazon/e-commerce site

Hybrid Data Systems

Data

Structured
(e.g. ads, purchases, product table)

Semi-structured
(e.g. user profile, web site activity)

Unstructured
(e.g. images, videos, docs)

DB Service

Structured
Relational

Semi-structured

Unstructured

Algorithms

JOINs, Aggregates
Indexing, Map-Reduce
Hashing, Sorting

Language/Tools

Hybrid future
- Built on same Lego blocks
- Past: SQL → noSQL → newSQL
- Now: hybrid SQL + pandas/ML

Example: Alibaba's data analysis design
1. How to scale to large data sets?
   - Is data relational, or unstructured or ...?
   - Is data in Row or Column store?
   - Is data sorted or not?

2. How do we organize search values?
   - E.g., Hash indices, B+ trees

3. How to JOIN multiple datasets?
   - E.g., SortMerge, HashJoins
Big Scale Lego Blocks

Roadmap

Primary data structures/algorithms

**Hashing**
- **HashTables**
  - $(hash_i(key) \rightarrow location)$

**Sorting**
- **BucketSort, QuickSort**
- **MergeSort**
- **MergeSortedFiles**
  - **MergeSort**
Sorting of relational T with N pages

\[ \sim 2N \left( \log_B \frac{N}{2(B + 1)} \right) + 1 \]

Sort N pages with B+1 buffer size

(vs n log n, for n tuples in RAM. Negligible for large data, vs IO -- much, much slower)

~ 2 N

Sort N pages when N \( \sim= B \)

(because \( \log_{0.5} 0.5 < 0 \))

~ 4 N

Sort N pages when N \( \sim= 2 \times B^2 \)

(because \( \log_B B = 1 \))

SortMerge and HashJoin for R & S

\[ \sim 3 \times (P(R) + P(S)) + \text{OUT} \]

Where P(R) and P(S) are number of pages in R and S, when you have enough RAM

\[ \sim 1 \times (P(R) + P(S)) + \text{OUT} \]

For SortMerge, if already sorted

For HashJoin, if already partitioned
1. Quick recap of scale

2. A full systems example putting things together
   ▶ How to compute data sizes?
   ▶ Execute a large join?
   ▶ Build an index
Billion products

User searches for “coffee machine”

Product recommendations

Customers who viewed this item also viewed these products

- Dualit Food XL1500 Processor
  - $580
  - Add to cart

- Kenwood kMix Manual Espresso Machine
  - $250
  - Select options

- Weber One Touch Gold Premium Charcoal Grill - 57cm
  - $225
  - Add to cart

- NoMU Salt Pepper and Spice Grinders
  - $3
  - View options
Counting popular product-pairs

Story: Amazon/Walmart/Alibaba (AWA) want to sell products

1. AWA wants fast user searches for product
2. AWA shows ‘related products’ for all products so users can explore
   - Using collaborative filtering (‘wisdom of crowds’) from historical website logs.
   - Each time a user views a set of products, those products are related (co-occur)

⇒ Goal: compute product pairs and their co-occur count, across all users

Data input:
- AWA has 1 billion products. Each product record is ~1MB (descriptions, images, etc.).
- AWA has 10 billion UserViews each week, from 1 billion users. Stored in UserViews, each row has <userID, productID, viewID, viewTime>. 
User search for product
1. Lookup Product Index
2. Lookup Products image
3. Lookup CoOccur Index
[1] + [2] + [3] < 100 msecs??
4. Capture user browsing info to UserViews

N_{Product} = ?
M_{Product} = ?
N_{CoOccur} = ?
M_{CoOccur} = ?
### Counting in RAM (pre-CS145)

**Counting product views for billion product PAIRs**

UserViews(UserId, ProductID, …, …)

<table>
<thead>
<tr>
<th>ProductID</th>
<th>ViewCount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nespresso Coffee</td>
<td>301</td>
</tr>
<tr>
<td>Bread maker</td>
<td>24597</td>
</tr>
<tr>
<td>Kenwood Espresso</td>
<td>22</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

CoOccur(ProductID1, ProductID2, count)

<table>
<thead>
<tr>
<th>ProductID1</th>
<th>ProductID2</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nespresso Coffee</td>
<td>Bread Maker</td>
<td>301</td>
</tr>
<tr>
<td>Bread maker</td>
<td>Kenwood Espresso</td>
<td>24597</td>
</tr>
<tr>
<td>Kenwood Espresso</td>
<td>Bike</td>
<td>22</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Algorithm**: For each user, product $p_i$, $p_j$

\[
\text{CoOccur}[p_i, p_j] += 1
\]
Counting product views

<table>
<thead>
<tr>
<th>Input size (4 bytes for user, 4 bytes for productid)</th>
<th>~1Bil * [4 + 4] = 8 GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output size (4 bytes for productid, 4 bytes for count)</td>
<td>~1Bil * [4 + 4] = 8 GB</td>
</tr>
</tbody>
</table>

Trivial

Counting product pair views

Plan #1: P * P matrix for counters in RAM (4 bytes for count)
- RAM size = 1 Billion * 1Billion * 4 = 4 Million TBs
- [Note: We’ll keep counter(p\textsubscript{i}, p\textsubscript{j}) and counter(p\textsubscript{j}, p\textsubscript{i}) in these examples. Why? We want to lookup cooccur counters both ways]

Plan #1 (on disk): Let OS page into memory as needed
- Worst case #1 > 100 million years
  (if you seek to update each counter)
Counting popular product-pairs

Your mission: Design an efficient system to compute co-occur counts on Sundays from weekly logs and produce a CoOccurCount table <productID, productID, count>

1. AWA’s data quality magicians recommend
   - (a) keep only top billion popular pairs, and (b) drop pairs with co-occur counts less than million.
   - (c) Also, assume users view ten products on average each week (User is interested in ~10 products/week, not 1000s).

2. For simplicity, SortedUserViews is stored sorted by <userID, productID>.
   - You can sequentially scan the log and produce co-occurring product pairs for each user. In other words, output (p_i, p_j) if a user viewed products p_i and p_j.
   - This “stream” of tuples (TempCoOccur) may then be (a) stored on disk or (b) discarded after updating any data structures.
Write a query like

```sql
SELECT TOP(..)...
FROM SortedUserViews v1, SortedUserViews v2
WHERE ...
GROUP BY v1.productId, v2.productId
HAVING count(*) > 1 million
```
Optimize, Evaluate design plans

Build Query Plans

1. For SFW, Joins queries
   b. Pre-build an index? B+ tree, Hash?

2. What statistics can I keep to optimize?
   a. E.g. Selectivity of columns, values

Cost in I/O, resources? To query, maintain?
Plan#1: With 1 machine, use RAM to count (Cost = 25B$ or > 100 million years).

Plan#2: With 1 machine

Plan 2
1. Scan SortedUserViews. For each user, **append** \(<p_i, p_j>\) to a file TempCoOccurLog if the user has viewed \(p_i\) and \(p_j\). (i.e., produce per-user co-occur product pair. **Append to log ⇒ No seek...**)
2. Externally sort TempCoOccurLog on disk, so identical product pairs are adjacent to each other in the sorted file
3. Scan sorted TempCoOccurLog. With a single pass, you can count co-occur pairs. Drop co-occur pairs with < 1 million.

<table>
<thead>
<tr>
<th>Nespresso</th>
<th>Iphone</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(\ldots)</td>
<td>(\ldots)</td>
<td></td>
</tr>
<tr>
<td>(\ldots)</td>
<td>(\ldots)</td>
<td></td>
</tr>
<tr>
<td>Nespresso</td>
<td>Iphone</td>
<td></td>
</tr>
<tr>
<td>(\ldots)</td>
<td>(\ldots)</td>
<td></td>
</tr>
<tr>
<td>(\ldots)</td>
<td>(\ldots)</td>
<td></td>
</tr>
<tr>
<td>Nespresso</td>
<td>Iphone</td>
<td></td>
</tr>
</tbody>
</table>

TempCoOccurLog
(After Step 1)

Sorted TempCoOccurLog
(After Step 2)

Count sorted TempCoOccurLog
(After Step 3)
Systems Design Example:

### Pre-design

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Size</th>
<th>Why?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProductId</td>
<td>4 bytes</td>
<td>1 Billion products ⇒ Need at least 30 bits ($2^{30} \approx 1$ Billion) to represent each product uniquely. So use 4 bytes.</td>
</tr>
<tr>
<td>UserID</td>
<td>4 bytes</td>
<td></td>
</tr>
<tr>
<td>ViewID</td>
<td>8 bytes</td>
<td>10 Billion product views.</td>
</tr>
<tr>
<td>Product</td>
<td>1 PB</td>
<td>1 Billion products of 1 MB each</td>
</tr>
<tr>
<td>SortedUserViews</td>
<td>240 GB</td>
<td>Each record is &lt;userID, productId, viewID, viewtime&gt;. Assume: we use 8 bytes for viewTime. So that’s 24 bytes per record. 10 Billion*24 bytes = 240 GBs.</td>
</tr>
<tr>
<td>CoOccur (for top 1 Billion)</td>
<td>12 GB</td>
<td>The output should be &lt;productId, productId, count&gt; for the co-occur counts. That is, 12 bytes per record ($4 + 4 + 4$ for the two productIdS and 4 bytes for count). To keep top billion product pairs (as recommended by AWA data quality), you need 1 billion * 12 bytes = 12 GBs.</td>
</tr>
<tr>
<td>TempCoOccurLog (assume: ~10 product views/user)</td>
<td>800 GB (12.5 M pages)</td>
<td># product pairs produced: 1 billion users * 10^2 = 100 billion Size @8 bytes/record (productId, productId) = 800 GBs</td>
</tr>
</tbody>
</table>
## Systems Design Example: Product CoOccur Plan #2

<table>
<thead>
<tr>
<th>Steps</th>
<th>Cost (time)</th>
<th>Why?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan SortedUserViews</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Append (&lt;p_i, p_j&gt;) to TempCoOccurLog</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Externally sort TempCoOccurLog on disk</td>
<td>~16,000 secs</td>
<td></td>
</tr>
<tr>
<td>(Assume sort cost is ~2N, where N is number of pages for table and B is number of buffers, and B ~ N)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scan TempCoOccurLog (sorted) and keep counts in CoOccur</td>
<td>~8000 secs</td>
<td></td>
</tr>
</tbody>
</table>
## Systems Design Example:

### Plan #2

<table>
<thead>
<tr>
<th>Steps</th>
<th>Cost (IO)</th>
<th>Why?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan SortedUserViews</td>
<td>4 M</td>
<td>240GB (4 M pages)</td>
</tr>
<tr>
<td>Append $&lt;p_i, p_j&gt;$ to TempCoOccurLog</td>
<td>12.5M</td>
<td>800 GB (12.5M pages)</td>
</tr>
<tr>
<td>Externally sort TempCoOccurLog on disk (Assume sort cost is ~2N, where N is number of pages for table and B is number of buffers, and B ~ N)</td>
<td>25M</td>
<td>IO cost is (appx) $= 2<em>N = 2</em>12.5M$</td>
</tr>
<tr>
<td>Scan TempCoOccurLog (sorted) and keep counts in CoOccur</td>
<td>12.5M</td>
<td>800 GB</td>
</tr>
</tbody>
</table>

Total IO cost = (4M + 12.5M + 25M + 12.5M) = 54M

Recall: Scan at 100 MBps, then time (secs) [assume, files are stored sequentially] $= (54M * 64 KB) /100 MBps = \sim 34.5K$ secs
Optimize, Evaluate
design plan 1, plan2, plan 3, ...

1. For SFW, Joins queries
   b. Pre-build an index? B+ tree, Hash?

2. What statistics can I keep to optimize?
   a. E.g. Selectivity of columns, values

Build Query Plans

Cost in I/O, resources? To query, maintain?
Build indexes with search key=productId. (Assume: CoOccur data may not be clustered)

CoOccur Index
<pid, pointer>

CoOccur
<pid, cooccurr_pid, count>

N_{CoOccur}: 167K pages (i.e. 1 billion rows * (4 + 8) bytes each in 64KB pages)

M_{CoOccur}: 1 billion rows * (4 + 4 + 4) bytes each (~167k pages)

For Index on Product data? [Recall: 1 billion tuples * 1 M B each = 1 PB]
- \( M_{Product} \): 1 PB/64 KB = 15.6 Billion pages
- \( N_{Product} \): <pid, pointer> = 167K pages
### Systems Design Example: Product CoOccur

- **B+ tree index**

**CoOccur Index**
- `<pid, pointer>`

**CoOccur**
- `<pid, cooccur_pid, count>`

**Data:** 1 billion records with 12 bytes each

- **Leaf:** `N = 167,000` pages
  - (h = 2, Each leaf can point up to 5460 search keys; Note: can grow up to 5460^2 before needing h=3)

- **Level 1:** <= 5460 pages, with 5460 ptrs each to next level

- **Root:** 1 page, with 5460 pointers

---

<table>
<thead>
<tr>
<th>N</th>
<th>167,000</th>
<th>From previous page</th>
</tr>
</thead>
<tbody>
<tr>
<td>How large is f?</td>
<td>~5460</td>
<td>4 bytes for productId + 8 bytes for pointers @ 64KB/page</td>
</tr>
</tbody>
</table>

\[ f \times 4 + f \times 8 \leq 64k \Rightarrow f \approx 5460 \]

Recall: We need a B+ tree of height \( h = \lceil \log_f N \rceil \)
Data Systems Design Example:

Product CoOccur

- \( N_{\text{Product}} = 167,000 \)
- \( M_{\text{Product}} = 15.6 \text{ Billion} \)
- \( N_{\text{CoOccur}} = 167,000 \)
- \( M_{\text{CoOccur}} = 167,000 \)

App/web browser

- Lookup Product Index
- Lookup Products image
- Lookup CoOccur Index

\([1] + [2] + [3] < 100 \text{ msecs}\)

User latency?

- \(~0.33M \text{ index Pages} \approx 21 \text{ GBs} \Rightarrow \text{Keep in RAM}~\)
Problem so far
- AWA’s product catalog is 1 billion items. AWA has 10 billion product views each week, from 1 billion users. Each log record stores <userID, productId, viewID, viewtime>

Consider 1000x Bigger problem!
- Product catalog is 1 trillion items. AWA has 10 billion product views. Rest stays same

⇒ What changes?
<table>
<thead>
<tr>
<th>Table</th>
<th>Size</th>
<th>Why?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProductId</td>
<td>8 bytes (vs 4 bytes)</td>
<td>1 trillion products ⇒ Need at least 40 bits (2^{40} \approx 1) Trillion to represent each product uniquely. So use 8 bytes (i.e. 64 bits).</td>
</tr>
<tr>
<td>UserID</td>
<td>4 bytes</td>
<td>“</td>
</tr>
<tr>
<td>UserViewsID</td>
<td>8 bytes</td>
<td>10 Billion product views.</td>
</tr>
<tr>
<td>Product</td>
<td>1000 PB</td>
<td>1 Trillion products of 1 MB each</td>
</tr>
<tr>
<td>Users</td>
<td>Unknown</td>
<td>Each record is (&lt;\text{userID, productID, viewID, viewtime}&gt;). Assume: we use 8 bytes for viewTime. So that’s 28 bytes per record. 10 Billion*28 bytes = 280 GBs.</td>
</tr>
<tr>
<td>SortedUserViews</td>
<td>280 GB (vs 240 GB)</td>
<td>The output should be (&lt;\text{productID, productID, count}&gt;) for the co-occur counts. That is, 20 bytes per record ((8 + 8 + 4) for the two productIDs and 4 bytes for count). To keep top billion product pairs (as recommended by AWA data quality), you need 1 billion * 20 bytes = 20 GBs.</td>
</tr>
<tr>
<td>CoOccur</td>
<td>20 GBs (vs 12 GB)</td>
<td># product pairs produced: 1 billion users * (10^2) = 100 billion</td>
</tr>
<tr>
<td>TempCoOccur (with UserSession assumption, of ~10 views/user)</td>
<td>1600 GB (vs 800 GB)</td>
<td>Size @16 bytes/record = 1600 GBs.</td>
</tr>
</tbody>
</table>

1000x larger catalog? < 2x increase in run time!
Popular Systems design pattern

1. Efficiently compute ‘batch’ of data (sort, hash, count)

2. Build Lookup index on result (b+ tree, hash table)

3. For ‘streaming’ data, update with ‘micro batches’

Popular problems

1. Related videos (youtube), people (Facebook), pages (web)

2. Security threats, malware (security), correlation analysis
1. How to scale to large data sets?
   ▶ Is data in Row or Column store?
   ▶ Is data sorted or not?

2. How do we organize search values?
   ▶ E.g., Hash indices, B+ trees

3. How to JOIN multiple datasets?
   ▶ E.g., SortMerge, HashJoins
Histograms & IO Cost Estimation
Optimization

Roadmap

1. For SFW, Joins queries
   b. Pre-build an index? B+ tree, Hash?

2. What **statistics** can I keep to optimize?
   a. E.g. Selectivity of columns, values
Example
Stats for spatial and temporal data

E.g. carpool or Traffic model
E.g. user traffic
Histograms

• A histogram is a set of value ranges ("buckets") and the frequencies of values in those buckets

• How to choose the buckets?
  • Equi-width & Equi-depth

• High-frequency values are very important (e.g., related products)
Example

How do we compute how many values between 8 and 10? (Yes, it’s obvious)

Problem: counts take up too much space!
What if we kept average only?

How much space do the full counts (bucket_size=1) take?

How much space do the uniform counts (bucket_size=ALL) take?

And Average Error?

E.g., uniform count = 3 (average)
Fundamental Tradeoffs

- Want high resolution (like the full counts)
- Want low space (like uniform)
- Histograms are a compromise!

So how do we compute the “bucket” sizes?
Equi-width

Partition buckets into roughly same width (value range)
Equi-depth

Partition buckets for roughly same number of items (total frequency)
Histograms

• Simple, intuitive and popular
• Parameters: # of buckets and type
• Can extend to many attributes (multidimensional)
Maintaining Histograms

- Histograms require that we update them!
  - Typically, you must run/schedule a command to update statistics on the database
  - Out of date histograms can be terrible!

- Research on self-tuning histograms and the use of query feedback
Compressed Histograms

One popular approach

1. Store the most frequent values and their counts explicitly
2. Keep an equiwidth or equidepth one for the rest of the values

People continue to try fancy techniques here *wavelets*, *graphical models*, *entropy models*, …
1. For SFW, Joins queries
   b. Pre-build an index? B+ tree, Hash?

2. What **statistics** can I keep to optimize?
   a. E.g. Selectivity of columns, values

Cost in I/O, resources?
To query, maintain?

Build Query Plans

Analyze Plans