Lecture 5 - ML in database systems

Section 1. Learning in a DBMS

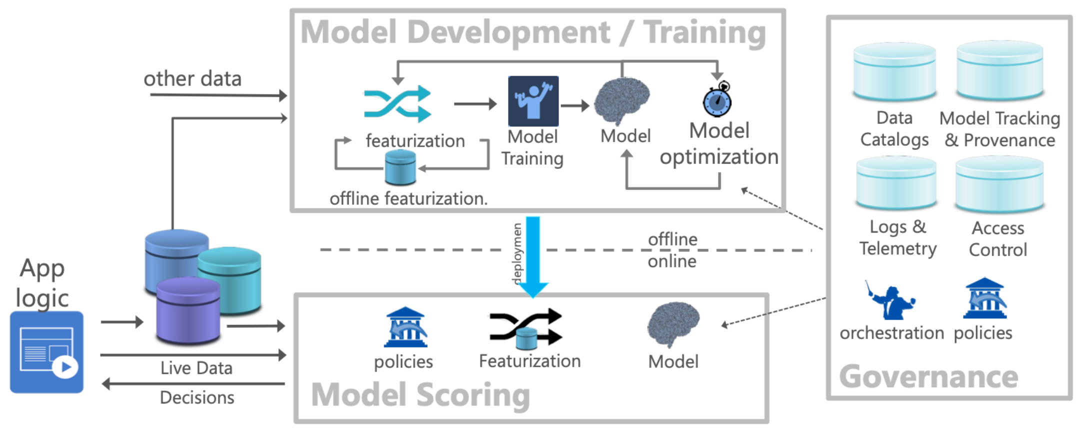


Figure 1: Flock reference architecture for a canonical data science lifecycle.

Source: Cloudy with High Chance of DBMS: A 10-year Prediction for Enterprise-Grade ML, CIDR 2020

Why ML in a Database System?

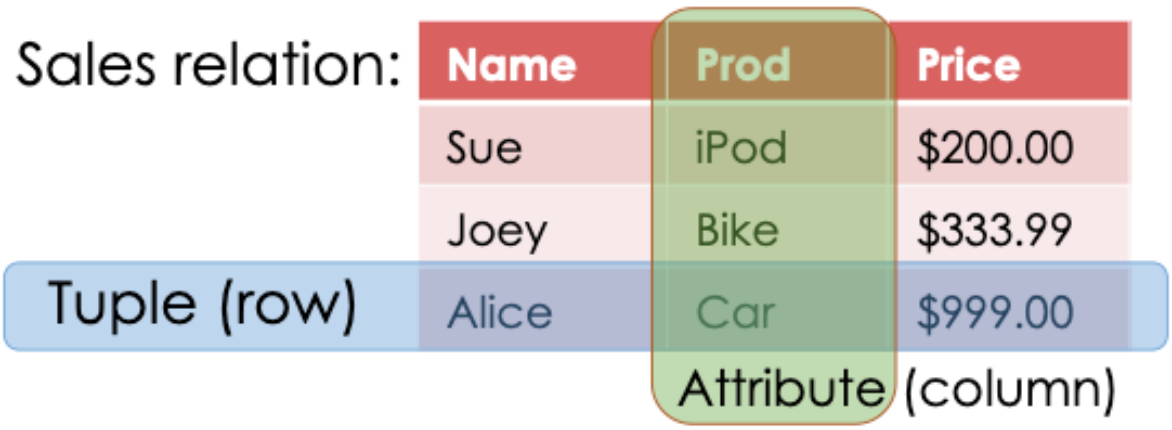
- Proximity to Data: minimize data movement
  - We want to avoid data duplication -> inconsistency
- Database systems are optimized for efficient access and manipulation of data:
  - Data layout, buffer management, indexing
  - Normalization can improve performance (We will see later)
  - Schema information can help in modeling/data validation
- Predictions with data: We can use trained models as user-defined predicates with data in the database
- Security: Data governance we can control who and what models have access to what data (we can leverage existing SLAs)

Challenges of Learning in Database

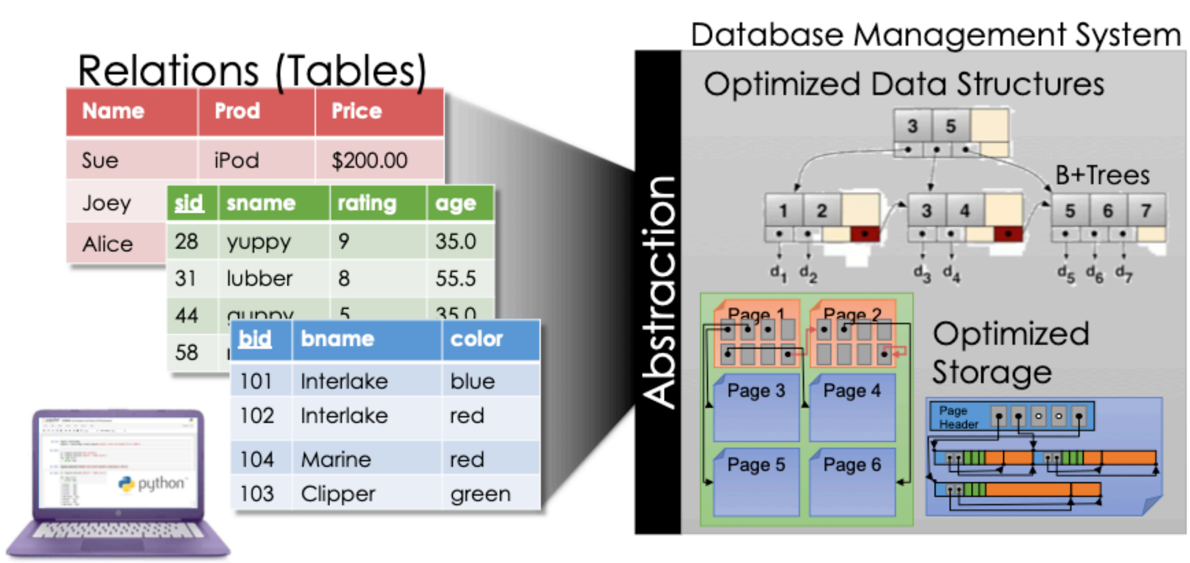
- **Abstractions:** The relational abstraction might not be the right one for learning algorithms
- **Access Patterns:** How does the ML algorithm access data? Sequentially, randomly, repeated scans

- **Cost Models and Learning:** Can the cost-optimizations of the DB system help?
- **New Data Types:** Images, video, models, how do we store them and manage them?

Section 2. Key Ideas in relational database management systems.



1) Logical data independence: Relational database system organize the data logically in relations (Tables). The ability to change the Conceptual (Logical) schema without changing the External schema (User View) is called logical data independence. For example, the addition or removal of new entities, attributes, or relationships to the conceptual schema or having to rewrite existing application programs.



2) Physical data independence: The ability to change the physical schema without changing the logical schema is called physical data independence. For example, a change to the internal schema, such as using different file organization or storage structures, storage devices, or indexing strategy, should be possible without having to change the conceptual or external schemas.

Database management systems hide how data is stored, The system can **optimize storage** and **computation** without changing applications.

The physical data layout/ordering of the data is determined by the system and the goal is to maximize performance.

3) Relational Algebra and Declarative Specification of Data Processing (SQL)

Declarative programming: Users just need to state what they want not how to implement it and how to get it.
Advantages of declarative programming: Enable the system to find the best way to achieve the result (optimization), more compact and easier to learn for non-programmers (?)

Challenges of declarative programming

- System performance depends heavily on automatic optimization
- Some languages may not be Turing complete (user-defined extensions)

Another big advantage of databases is: **out-of-core computation**

Premise: data does not fit in memory
Database systems are typically designed to operate on **databases larger than main memory**

- Algorithms must manage **memory buffers** and **disk**
  - Page level memory buffers
  - Sequential reads/writes to disk
- Understand **relative costs** of memory vs disk
- Core idea: bring part of the data in memory and operate on it

EXAMPLE: Grace hash join

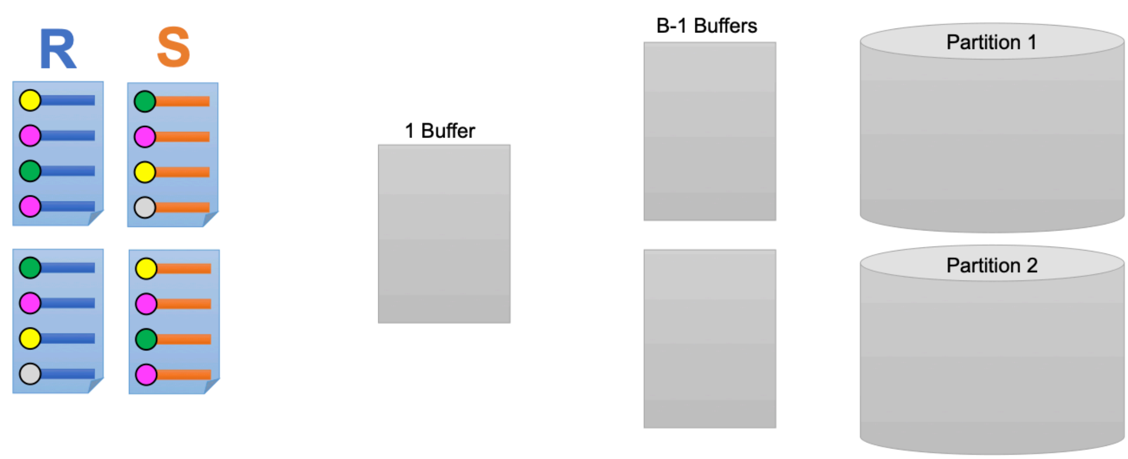
Grace Hash Join

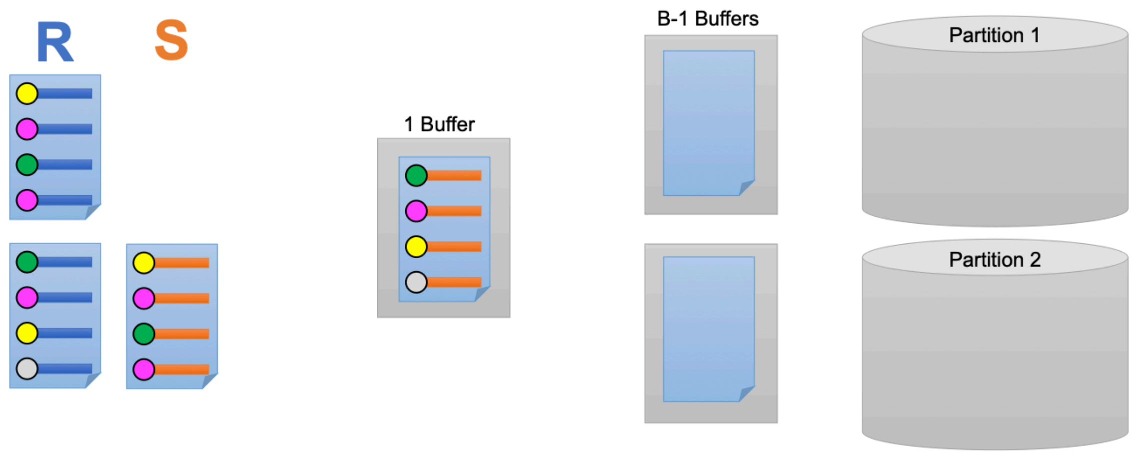
$$R \bowtie_\theta S = \sigma_\theta(R \times S)$$

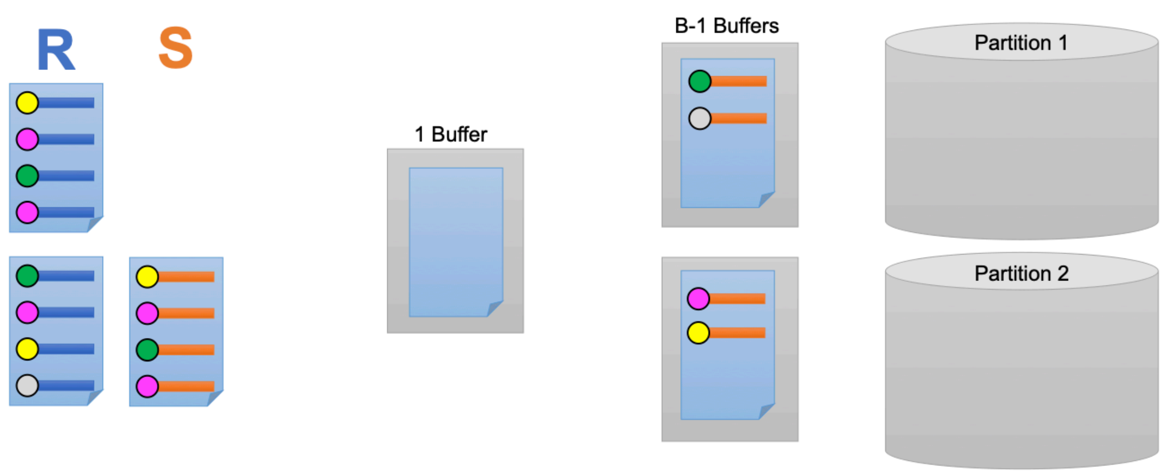
- Requires equality predicate θ:
  - Works for **Equi-Joins** & **Natural Joins**

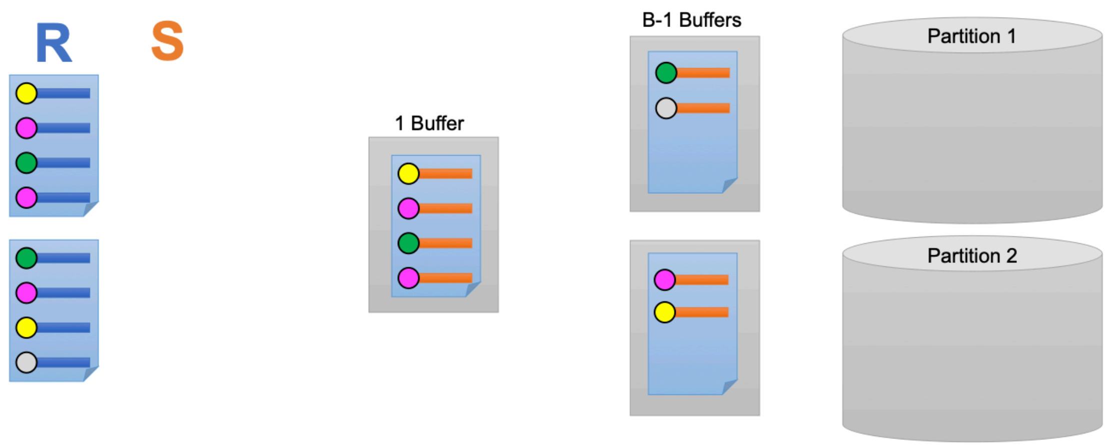
- Two Stages:
  - Divide **Partition** tuples from R and S by join key
    - all tuples for a given key in same partition
  - Conquer **Build & Probe** a separate hash table for each partition
    - Assume **partition** of smaller rel. fits in memory
      - Recurse if necessary…

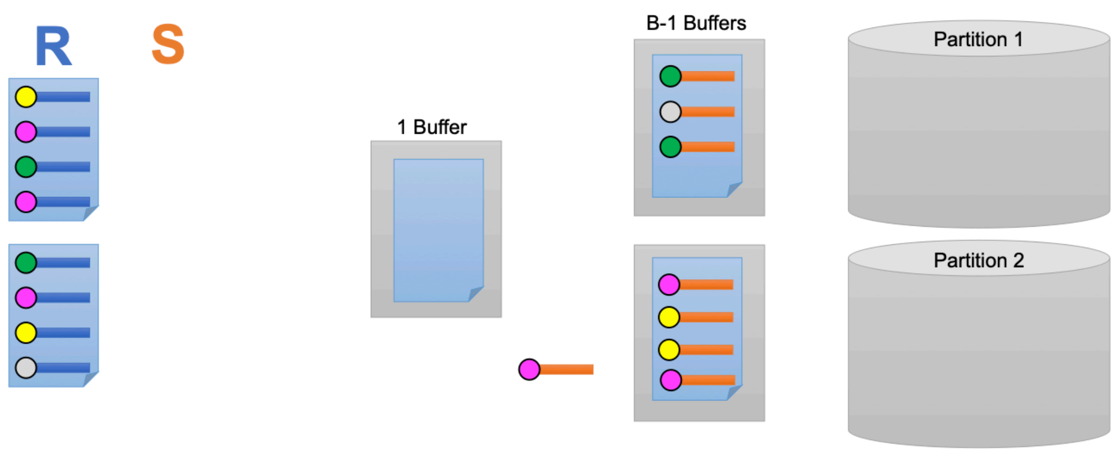
Grace Hash Join 1: Partition Phase

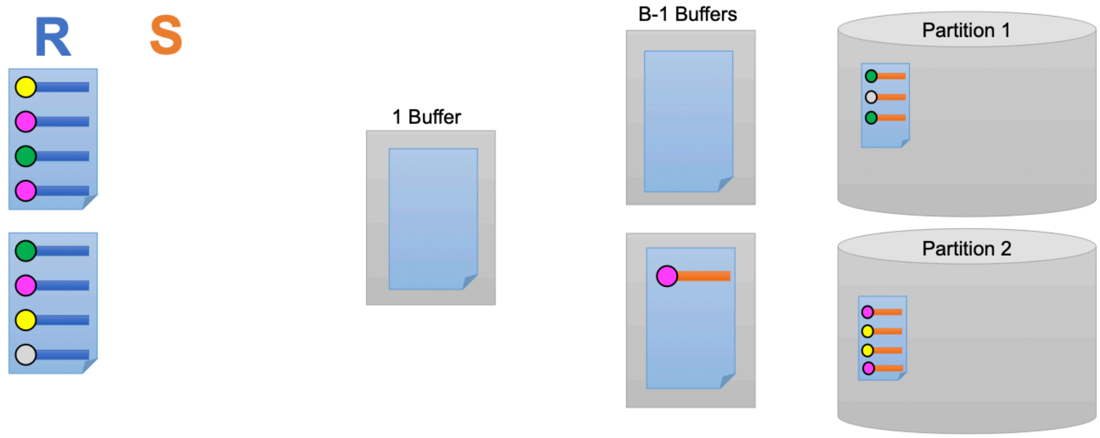



R
S
1 Buffer
B-1 Buffers
Partition 1
Partition 2

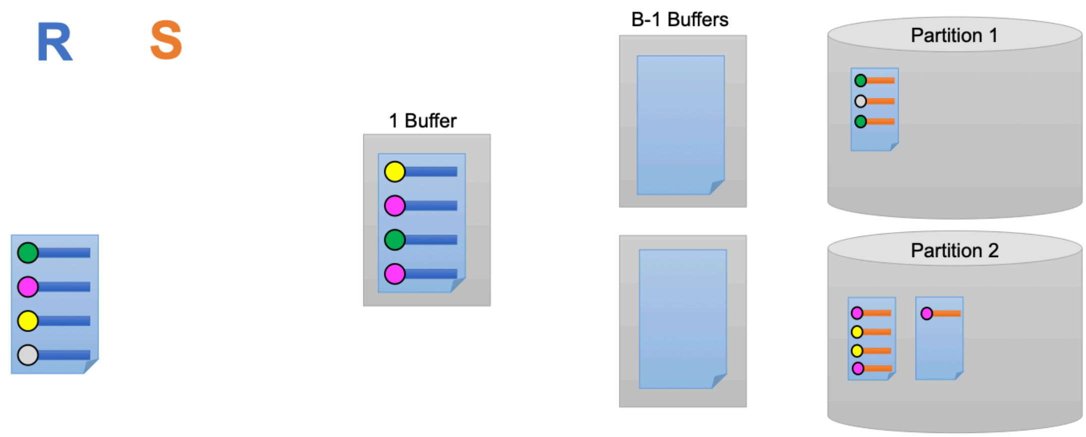

R
S
1 Buffer
B-1 Buffers
Partition 1
Partition 2


R
S
1 Buffer
B-1 Buffers
Partition 1
Partition 2

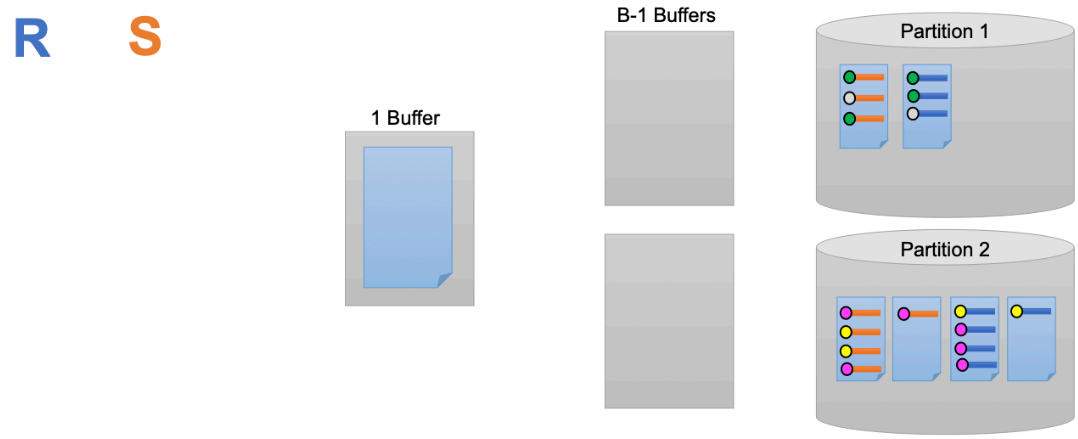

R
S
1 Buffer
B-1 Buffers
Partition 1
Partition 2



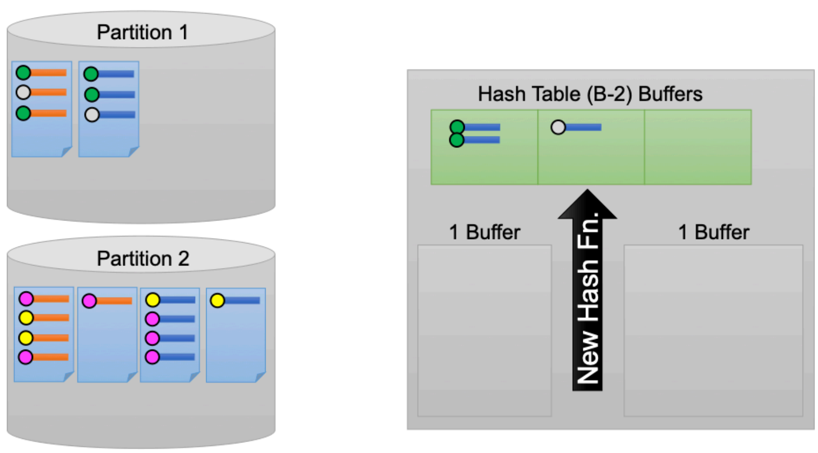
R
S
1 Buffer
B-1 Buffers
Partition 1
Partition 2

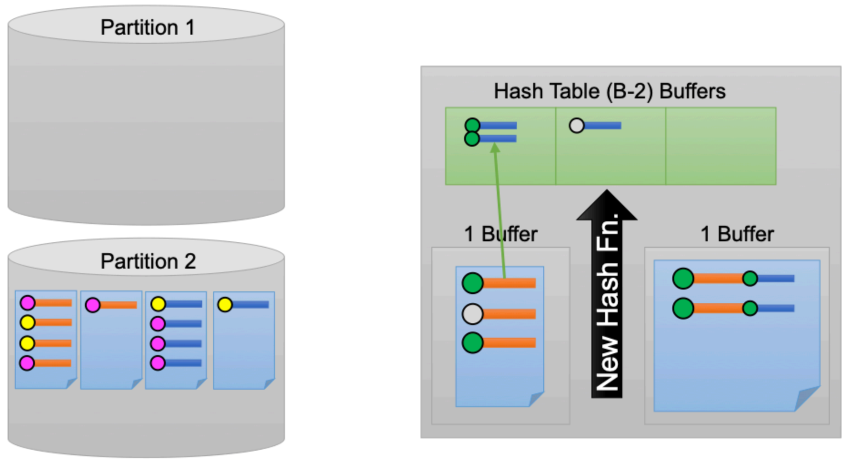


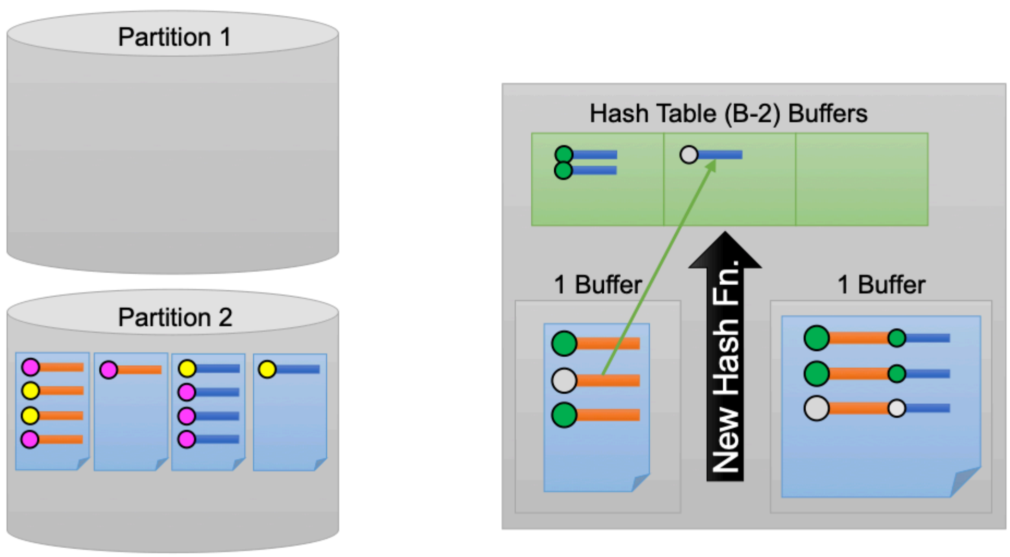
R
S
1 Buffer
B-1 Buffers
Partition 1
Partition 2

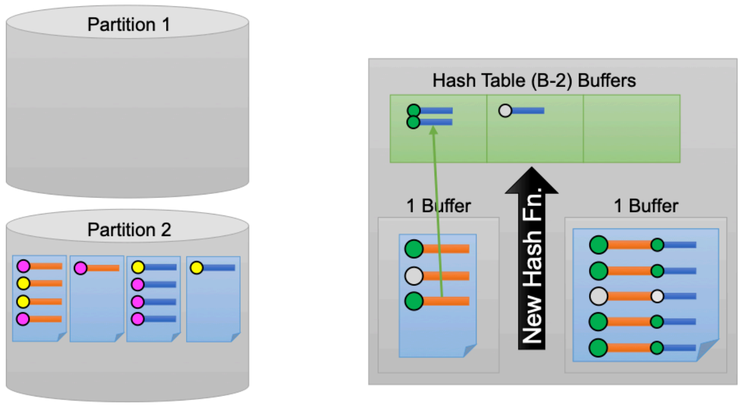


Grace Hash Join 2: Build and Probe

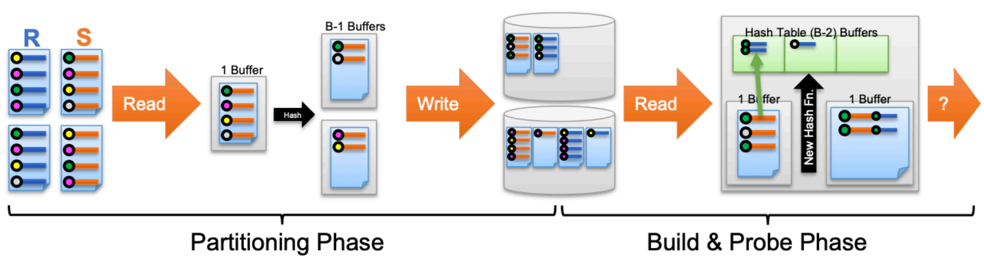



Partition 1
Partition 2
Hash Table (B-2) Buffers
1 Buffer
New Hash Fn.
1 Buffer


Partition 1
Partition 2
Hash Table (B-2) Buffers
1 Buffer
New Hash Fn.
1 Buffer



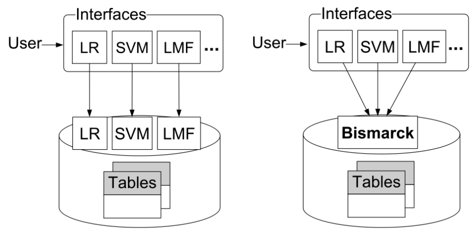
Cost of Hash Join



- Partitioning phase: read+write both relations
  $\Rightarrow$ 2([R]+[S]) I/Os
- Matching phase: read both relations, forward output
  $\Rightarrow$ [R]+[S]
- Total cost of 2-pass hash join = 3([R]+[S])

Section 3. Examples of ML and DB integration.

Reading: https://www.cs.stanford.edu/people/chrismre/papers/bismarck.pdf



Current In-RDBMS Analytics   Bismarck In-RDBMS Analytics

Analytics Task	Objective
Logistic Regression (LR)	$\sum_i \log(1 + exp(-y_i w^T x_i)) + \mu\|\vec{w}\|_1$
Classification (SVM)	$\sum_i (1 - y_i w^T x_i)_+ + \mu\|\vec{w}\|_1$
Recommendation (LMF)	$\sum_{(i,j)\in\Omega} (L_i^T R_j - M_{ij})^2 + \mu\|L, R\|_F^2$
Labeling (CRF) [48]	$\sum_k \left[\sum_j w_j F_j(y_k, x_k) - \log Z(x_k)\right]$
Kalman Filters	$\sum_{t=1}^T \|\|Cw_t - f(y_t)\|\|_2^2 + \|\|w_t - Aw_{t-1}\|\|_2^2$
Portfolio Optimization	$p^T w + w^T \Sigma w$ s.t. $w \in \Delta$

Many ML techniques (mostly generalized linear models) can be reduced to **mathematical programming** and there is a single solver (**Incremental Gradient Descent**) that fits existing database system abstractions (**User Defined Aggregates**).

```
LR_Transition(ModelCoef *w, Example e) { ...
  wx = Dot_Product(w, e.x);
  sig = Sigmoid(-wx * e.y);
  c = stepsize * e.y * sig;
  Scale_And_Add(w, e.x, c); ... }
```

```
SVM_Transition(ModelCoef *w, Example e) { ...
  wx = Dot_Product(w, e.x);
  c = stepsize * e.y;
  if(1 - wx * e.y > 0) {
    Scale_And_Add(w, e.x, c); } ... }
```

```
CREATE AGGREGATE bismarck (…){
  initialize(args) → state:
     randomly initialize model weights
  transition(state, row) → state:
     single gradient update
```

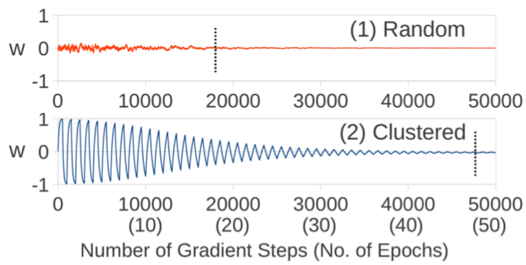
$$w^{(k+1)} \leftarrow w^{(k)} - \alpha_k \nabla L\left(\text{row}, w^{(k)}\right)$$

```
  terminate(state) → result
     return current model for epoch
  merge(state, state) → state
     used for parallel model averaging
}
```

- State contains:
  - Model weights, k, …
- Invoked repeatedly
  - Once per epoch
  - Bismarck stored procedure
- Termination cond.
  - Similar to IGD

Data Ordering Issues

- Data indexed/clustered on key feature or even the label
  - **Example:** predicting customer churn → data is partitioned by active customers and cancelled customers
    - Why?
- May slow down convergence:



Random Order

Sorted by **label**

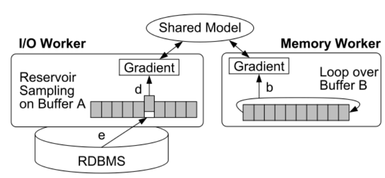
Data Order Solutions

Shuffle data

- **on each epoch** (pass through data): Closest to stochastic gradient alg.
  - Expensive data movement and duplication
- **Once:** good compromise but requires data movement and dup.

Sample data

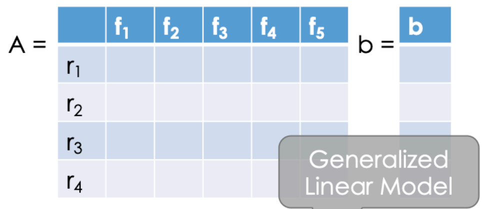
- **single reservoir sample per pass**
  - Train on less data per scan → slower convergence
- **multiplexed reservoir sampling**
  - Concurrently training on sample and raw data streams



Section 4. Reading on Workload optimization

Problem Formulation

Data



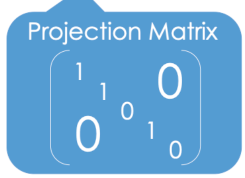
Solve (within ϵ of optimum)

$$x_t^* = \arg\min_{x \in \mathbb{R}^d} \sum_{i \in R_t} L\left((A\Pi_{F_t} x)_i, b_i\right)$$

For each t:

R_t : set of rows

F_t : set of cols



- Solve **multiple problems** for **subsets** of **rows** and **columns** of original data
- Block consists of:
  - Loss functions L
  - **Set of Sets** of Rows / Columns
  - Accuracies ϵ
- Explore **optimizations** targeted at solving the **related problems**
  - ***Materialization, Sampling, Compute reuse***

Optimization: Lazy vs Eager Materialization

- **Lazy Materialization:** construct each feature table as it is needed from raw data
- **Eager Materialization:** precomputes the superset of columns (features) and then projects away what is not needed for each optimization task
- **Tradeoffs**
  - **Lazy** → Higher computational cost, less storage overhead
  - **Eager** → Less compute, greater storage overhead