Spreadsheets and Dataframes

There's order in disorder: A tale of three data models

Some History:

- Codd's relational model: 1969; System R: 1974
  - from the "database" community
- Spreadsheets: first prototype: 1969; first implementation for microcomputers: 1979
  - from the accounting community
  - ledgers that present the results of calculations along with data
- Dataframes: first included in S: 1990 -> R -> Python: 2008
  - from the statistics community
  - generalizing from matrices (homogeneous) to heterogeneous types
  - one observation per row, one variable per column

Order in a relational database

- ORDER BY: typically done "at the end"
- Implicitly done for certain types of operators, e.g., GROUP BY, and by certain physical implementations, e.g., sort-merge join -- recall interesting orders for Selinger's dynamic programming algorithm
- WINDOW functions
  - Only introduced in SQL as part of SQL:2003 standard

WINDOW functions

- Rarely taught in database classes, because it is so new
- Provides a way to operate on order as a first-class citizen
  - Look up "nearby" rows, based on some grouping and ordering
  - But you don't need to "squish" down to a single value per group
- Central to many analytics tasks
  - `SELECT salary, AVERAGE (salary) OVER () FROM employees;`
  - `SELECT salary, RANK () OVER (PARTITION BY dept ORDER BY salary) FROM employees`

- `SELECT sensorid, value, AVERAGE (value) OVER (PARTITION BY sensor ORDER BY time ROWS 5 PRECEDING AND CURRENT ROW) AS previous_average`

- can also do cumulative sum, etc.
- Conceptual flow: partition -> order -> window -> aggregate
- A blocking operation much like GROUP BY & SORT

```
<window or agg_func> OVER (
  [PARTITION BY <…>]
  [ORDER BY <…>]
  [RANGE BETWEEN <…> AND <…>])
```

1. compute a function
2. over a particular window
3. where the window tuples are ordered like so
4. and take this particular frame of interest in the window

NB: Conceptually evaluated after `GROUP BY/HAVING`

4. RANGE entation 3.5, 4.2.8, 9.22

```
<window or agg_func> OVER (
```

1. Compute a function

```
  [PARTITION BY <…>]
```

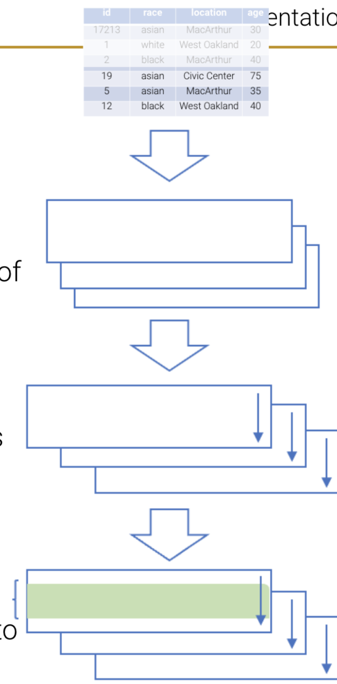
2. over a particular window
(Partition into windows the larger group of rows that the current row is part of)

```
  [ORDER BY <…>]
```

3. where the window tuples are ordered
(Define the way this partition (window) is laid out)

```
  [RANGE BETWEEN <…> AND <…>])
```

4. and take this particular frame of interest in the window
(the subset within this ordered partition to compare against)



This sort of stuff is very natural in spreadsheets

- Once you impose an order, can perform windowed aggregation, e.g., `Bi = SUM ($A$1: Ai)` for cumulative sum

Other reasons why order helps:

- You can refer to data by position
- Order often has meaning, especially when you're presenting results of analysis
- Can inspect results in order as operations are performed (debugging)

Ways order doesn't help:

- Constraint that needs to be met after every operation
- Maintaining this is expensive
- Often leads to tight coupling between logical and physical representations
- Brittle, buggy: >50% of spreadsheets contains mistakes
    - e.g., if your formula is referencing the first 28 rows, but doing so because that corresponds to all of February, now, when you have a leap year, that formula is no longer correct

Let's talk about spreadsheets

- Super popular, 1B users!
- In 2015, when we started this work, largely not an interesting topic of study
- Bakke et al. SIGMOD 2016: allowing hierarchical representations within spreadsheets, admitting GROUP BY-like expressions
- Joe and colleagues did some work on spreadsheets in 1999 "Scalable Spreadsheets for Interactive Data Analysis.", workshop paper.
    - Spreadsheets was more "in the background" rather than the foreground
    - Emphasis ended up being
        - Online aggregation for presenting early results of aggregates, approximately
        - Online dynamic reordering for prioritizing the generation of what the user may be currently seeing (coupled with something like eddies)
- Mappings between spreadsheets and databases
    - XLOOKUP - a kind of foreign key join
    - formulae = materialized views
    - typical aggregate functions, plus those with IF, e.g., SUMIF - much like SUM + a WHERE clause

A brief HCI aside: direct manipulation

Direct manipulation (coined by Shneiderman in 1982) user interfaces have three properties:

- **Continuous representations** of the objects and actions of interest;
- **Physical actions** instead of complex syntax; and
- **Rapid, incremental, reversible operations** whose effect on the object of interest is **immediately visible**.

Key benefits:

- Novices can learn via demonstration; experts can define new features/functions for rapid work.
- Users experience less anxiety because the system is comprehensible & actions can be reversed.
- Users gain confidence and mastery because they are initiators of actions, they feel in control, and system responses are predictable.

But spreadsheets don't actually scale (2020 paper)

- don't go beyond 1M rows
- operate entirely in main-memory
- no real query/storage optimization
  - each formula evaluated one at a time, including XLOOKUP (a kind of foreign key joins)
  - no subexpression elimination
  - no indexes
  - no careful layout of data
- I like to joke that spreadsheets invented the n^2 log n sort. If you aren't careful.

Our goal in the ICDE 2018 paper was twofold:

- build a more scalable spreadsheet
- build a spreadsheet frontend to a database -> not as interesting

Goal A: Building a more scalable spreadsheet boils down to representation and access

Representation question:

- How do you efficiently represent spreadsheets?
- The paper does this by identifying tabular and non-tabular regions:
  - tabular regions, store in some row or columnar format
  - non-tabular regions, store in k-v format

- Gains in storage compared to just tabular or just non-tabular (which is what is done right now), but also reduced formula computation costs, because related data is present "close by"
- algorithm that recursively divides the spreadsheet, much like KD trees

Access question:

- How do you efficiently maintain position during updates, e.g., adding/deleting rows?
- First alternative: store position directly
  - downside: cascading update O(n)
  - OTOH can use a standard B+tree on position for locating kth record O(log n)
- Second alternative: monotonically increasing proxies 0, 10, 20, 30, ...
  - no cascading updates (to a limited extent)
  - downside: mappings are lost, so O(n) lookups
- Counted B+ trees
- Each node also stores the count of nodes below it
- Updates and lookups in O(log n)

But: barely scratching the surface. Lots more to be done, from our group

- we built an asynchronous execution engine for spreadsheet formulae SIGMOD 2019
- compression for spreadsheet formula networks ICDE 2023
- frontend: very hard to make sense of large spreadsheets VLDB 2021

Let's talk about dataframes

- central to data analysis and data science
  - pandas often cited as the reason for python's popularity
  - called the most important tool in data science
  - used for everything ranging from data cleaning to even primitive ML, and even more especially in combination with ML libraries
- 500+ functions, allowing you to do anything you want to your data
  - Lots of redundancy: many ways to do the same thing (1700x change)
- We identified that pandas was being used to operate on very large datasets and was breaking down
  - Much like spreadsheets, often would OOM
  - Very inefficient w/ memory would, make multiple copies

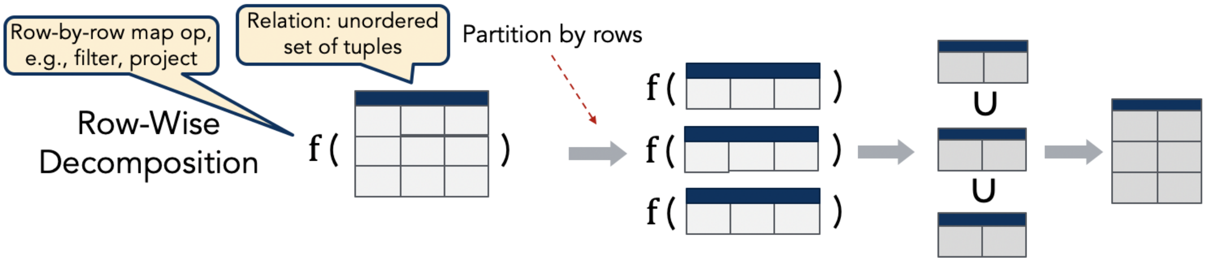
    - Each operator executed by itself, in entirety, no optimization
    - Eager metadata maintenance
- So we started building Modin, a parallel dataframe system, architected as a "drop-in" replacement for pandas
- Here, the papers came after the system was already gaining traction in the OSS community
- Started a company, which was sold to Snowflake
- Now has 1M+ downloads a month

Let's talk about the 2020 paper that crystallized the dataframe data model and algebra

- Lots of math in the paper, but at the highest level, a dataframe is a four-tuple (Array, Row Labels, Column Labels, Types)
- Unlike relations:
    - ordered along both rows and columns
    - rows are named
    - types can be unspecified after operations, and may need to be induced (as in spreadsheets)
- From an algebraic standpoint
    - output could have arbitrary schema that depended on data
        - e.g. one-hot encoding or pivot, or dropNA along columns
        - simply a no-no for
    - data was equivalent to metadata, so we could move information from the labels to the data and vice-versa
- We also boiled all the operators down to a small number of operators (we refined this in the second 2021 paper), that included
    - ordered versions of relational operators along both rows and columns
        - e.g., filter along columns
    - to/from labels
    - transpose

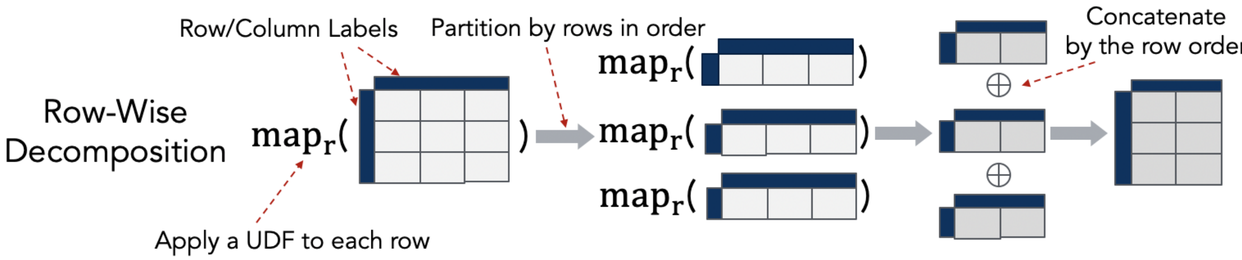
Next, how do we think about parallelizing the evaluation of dataframe operators?

- can we apply the same ideas from parallel databases, e.g., breaking an operation on an entire relation into those on partitions?
- two twists:
    - order and access



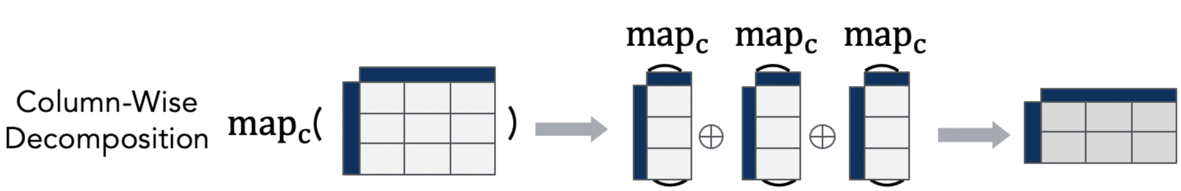
More complicated for blocking/binary ops,
e.g., sort, group by, joins
will focus on the simplest case for this talk

Decomposition Rules for Dataframes: Remember Order!

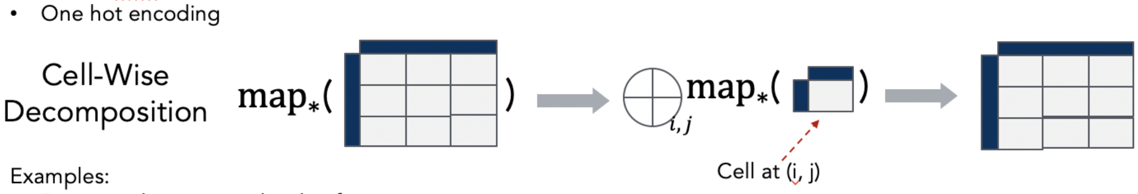


Examples:
- add a derived feature
- explode a row into multiple rows

Decomposition Rules for Dataframes: Remember Access!

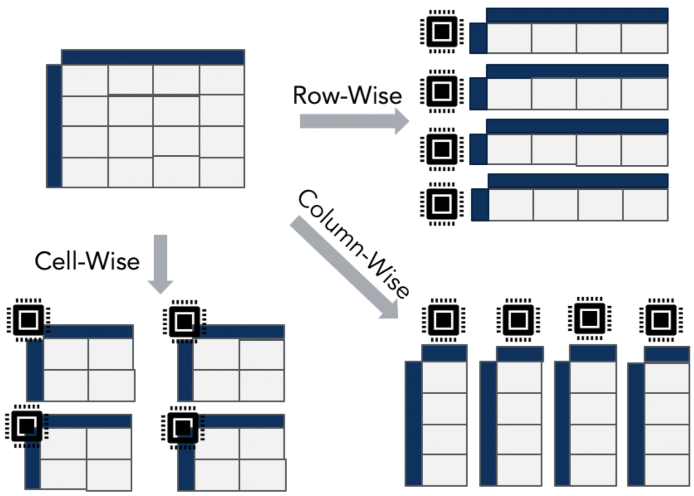


Examples:
- Fill the NaN values of each column based on a UDF
- One hot encoding



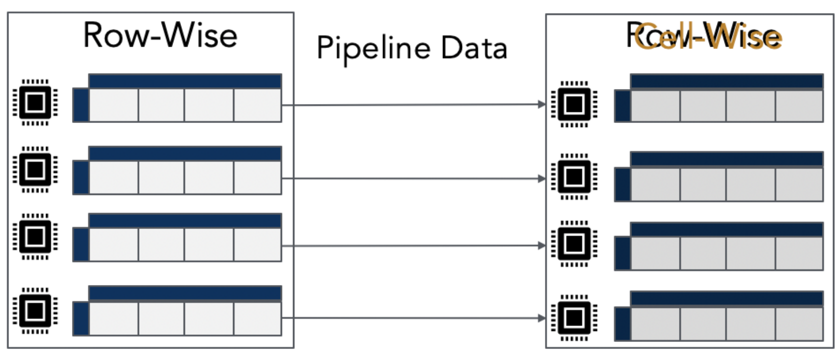
Examples:
- Regex replace across the dataframe

Applying Decomposition Rules for a Single Operator



Applying Decomposition Rules to a Chain of Operators

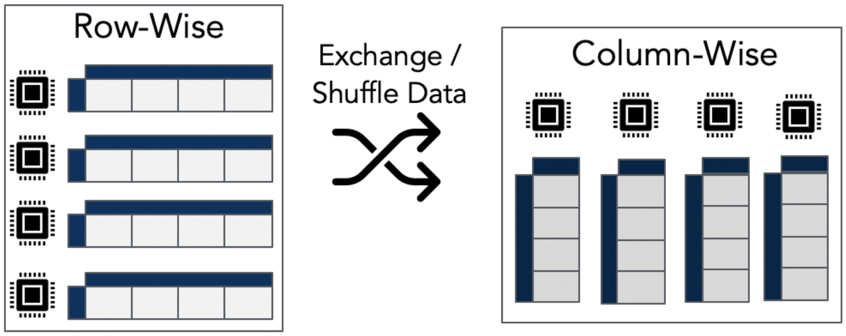
Need to decide communication between successive operators



Cell-wise decomposition is more flexible wrt its input than row/column-wise decomposition

Applying Decomposition Rules to a Chain of Operators

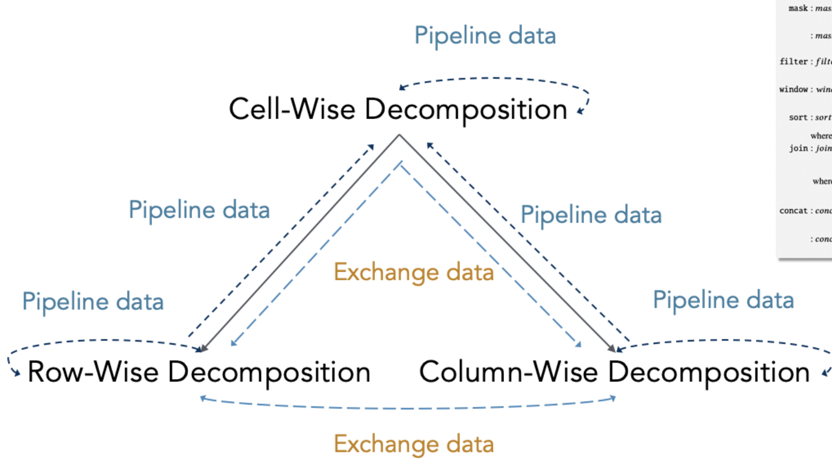
Need to decide communication between successive ops

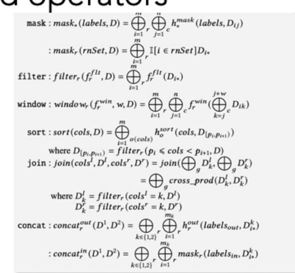


Exchanging data is much more costly than pipelining

Applying Decomposition Rules to a Chain of Operators

Deciding to pipeline or exchange data between two chained operators

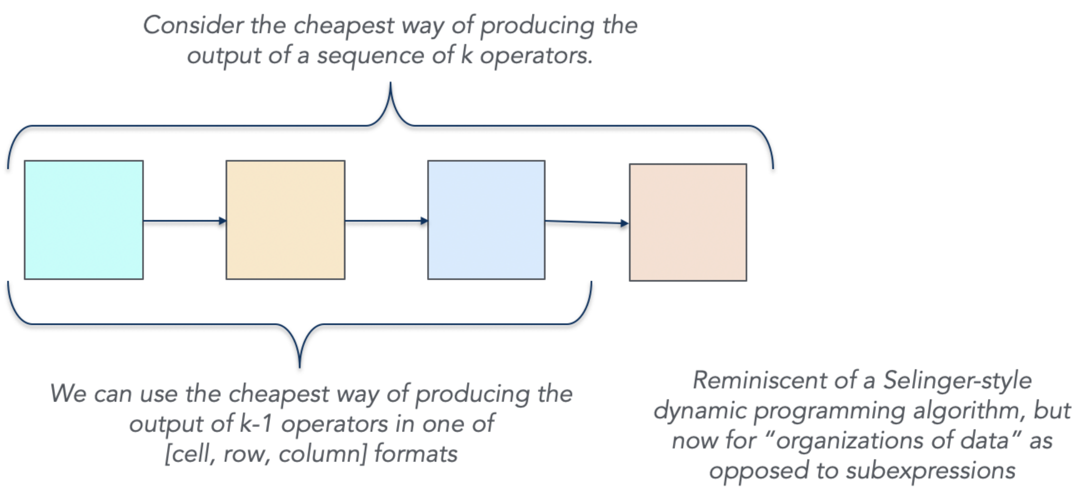




[actually more complex hierarchy, see paper!]

For a chain of ops how do we communicate in-between to minimize total latency?

Overall Optimization: Simple Optimal Substructure Intuition



Paper also discusses ways to lazily maintain metadata

Vision paper outlines a bunch of open, unaddressed questions - still yet to be done, much like spreadsheets

--

If time, precision, expressiveness, usability tradeoff

- SQL
- NL2SQL
- Keyword search in databases
- spreadsheets
- query builders
- query by example: microsoft access

Takeaways:

- two other popular data "rectangular" models, beyond relational
- both center on order, and also admit disorder - loose typing, structure
- both worthy of study, barely scratching the surface of each