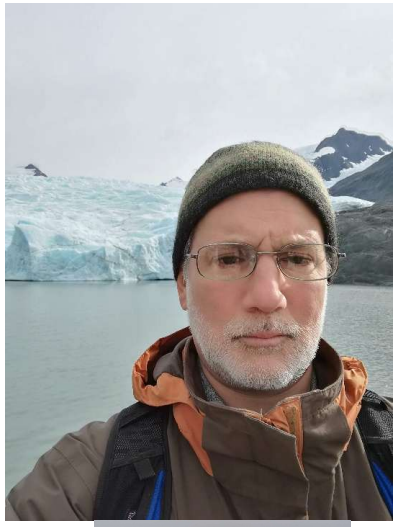
Cascades and the Scope system

MarcFr@microsoft.com

2026-Feb-9

Who is this guy?



Marc Friedman was the first Ph.D. graduate from the UW CSE's newly-formed database group in 1999. When you make pancakes, you always throw away the first one, right? Marc sat where you are now, alongside: Alon Levy (advisor), Zack Ives, Anhai Doan, Rachel Pottinger, Todd Millstein, and apologies to the other one or two people who came to database seminar that I cannot recall at this moment.

Marc studied planning and scheduling, database query optimization and data integration. He worked at 2 startups (both dead) doing data virtualization, and then back to query optimization at Microsoft, first in SQL Server, then MatrixDB (dead), then U-SQL(dead) and Scope. Marc lives walking distance from UW campus with his spouse, has one child in college, and is a big fan of ceramics and board games.

Terminology

- Distribution/distributed => could be referring either to data at rest or data-centric computation
- Partitioned => distributed
- Parallel => distributed
- Cascades => Microsoft SQL Server’s implementation thereof

Real problems with QO and Scope (1/2)

- [Scope] Data skew is a user issue … sometimes
  - Working despite data skew is a key goal of Scope
  - Sometimes the optimizer introduces it: SELECT DISTINCT * -- what to partition on?
- Plan stability over time
- Error compounds
- Cost given expected card ≠ expected cost
  - I.e. uncertainty is ignored
- Model of execution is just a model
- Resources
- How to hint the plan you really want
- There is no universal metric
- Cascades transforms are local. Global heuristics are hard.

Real problems with QO & Scope (2/2)

- You need to do adaptive OR interleaved OR feedback today
  - Expensive to build, maintain
- “Why that plan?” Is difficult to answer (even worse with above)
- Improving something (estimates, cost function, etc.) causes some regressions
- Tuning suggestions are easy; getting customers’ attention is not

What is Scope (1/3)

- One of ~10 database engines at Microsoft
- Big data engine
  - MaxColumnSize=MaxRowSize = 250MB
  - Page size = 250MB
  - We put a limit of 5PB on job inputs but customers have started hitting it
- Scope is 1/3 of the 'Cosmos' ecosystem, alongside Yarn++ resource mgr and cosmos store
- SQL-like language (also called 'Scope') with MAP/ARRAY/STRUCT
- C# types
- C# scalar expression language
- Full .Net library support
- Related systems: HIVE, Cassandra, BigTable, BigQuery, Spark, …

What is Scope: the big data domain

- The workload is analytics, data mining, ETL, data prep, billing, ML
- Lots of scanning, merging, shuffling, and custom code
- Extensible: Integration with C#, Python user defined things:
  - Operators: Extractors, Outputters, Processors, Combiners, Reducers
  - Scalar functions
  - Aggregate functions
  - Types
- Batch, i.e. job form factor. All data is cold. You want a batch to do everything while it has the data in flight. Hence multiple inputs, multiple outputs. 1 job = 1 script = 1 DAG.
- Lots of automated pipelines consisting of 10s of related jobs
- Price performance at scale is supreme
- Highly reliable execution despite every component being unreliable

What is Scope (3/3)

- Dryad execution model
- Reads/writes files, structured streams, and now delta tables
- Not available to the public, not open-source
- Used to also have interactive, streaming, and ML variants (all dead)

Structured streams

- We recommend all cosmos-internal data be in structured streams.
- A partitioned clustered index:

  OUTPUT "/out/x.ss" HASH CLUSTERED BY cust_id INTO 100 SORTED BY cust_id, order_id
  OUTPUT "/out/y.ss" RANGE CLUSTERED BY url ASC SORTED BY url
- Each partition is SORTED the same way.
- Stored as a conventionally-organized folder with a metadata file.
- Immutable (originally)
  - more /out/x.ss*

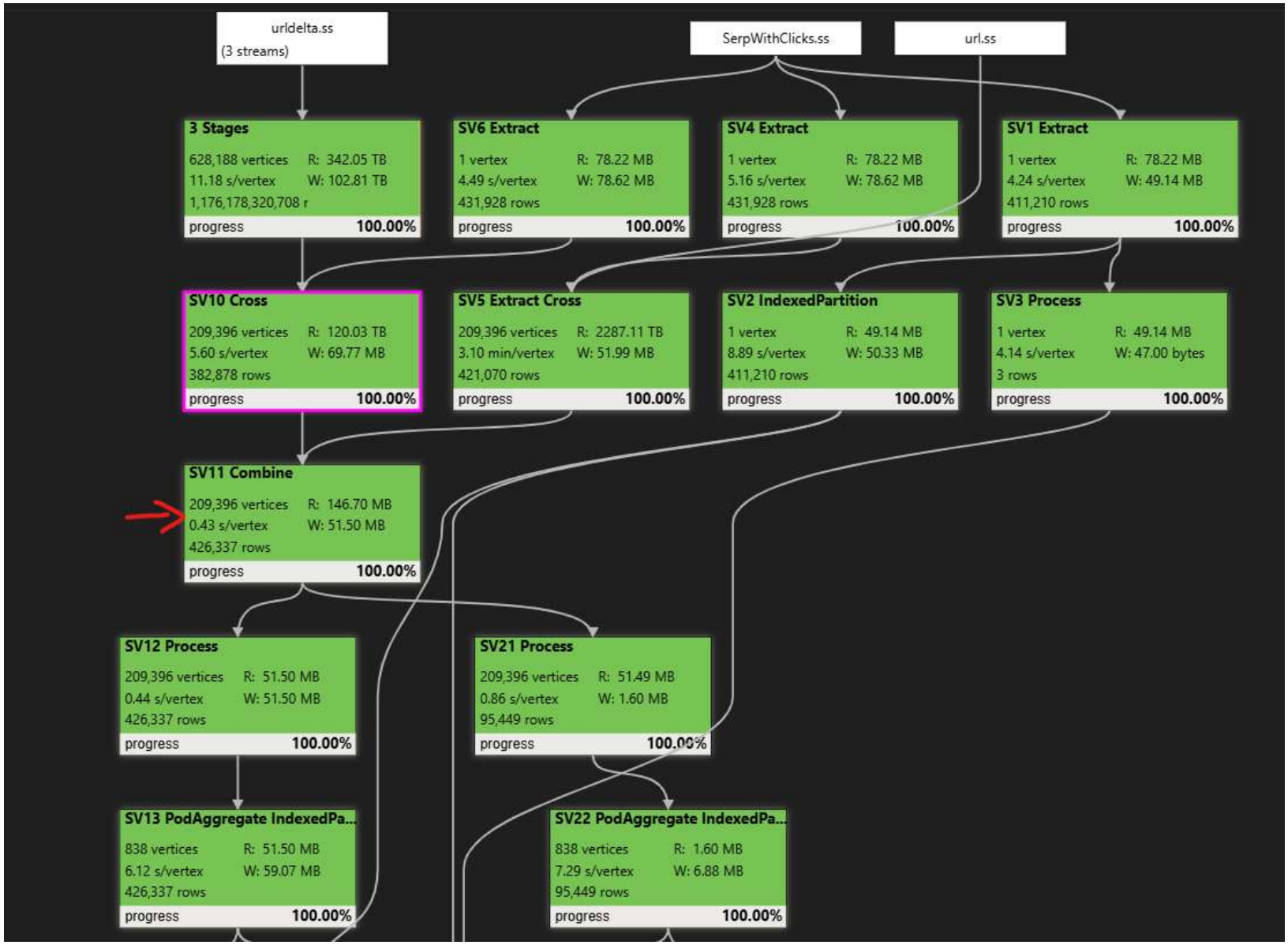
    x.ss // a folder
    x.ss.metadata // contains the guid
  - ls /out/x.ss/<guid>

    unit_0.data
    unit_1.data ...
    unit_99.data
- Limited to built-in types

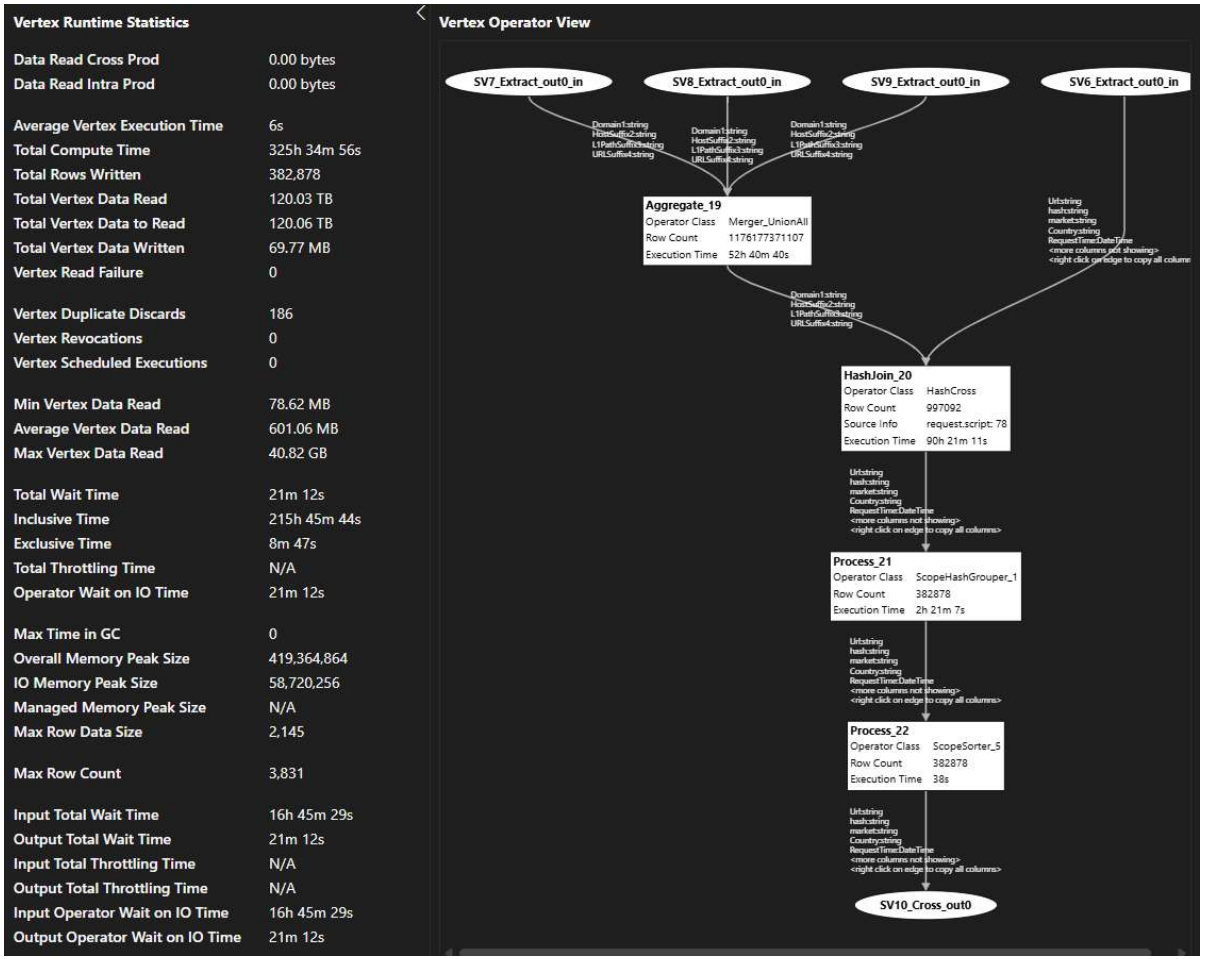
The Scope query optimizer

- Forked from the Microsoft SQL Server optimizer 2006
- Input and output are DAGs
- Shuffle is a physical operator
- Key goals
  - Minimize data movement (shuffling)
  - Shuffle placement
  - Choose DOP to trade off parallel (distributed) execution vs. overhead
  - Protection from data skew
  - Everything can be hinted
- More recently, learn from history
  - Automatic materialized views
  - Interesting columns, card, container size, operator memory, tokens

Job view


urldelta.ss
(3 streams)
SerpWithClicks.ss
url.ss
3 Stages
628,188 vertices R: 342.05 TB
11.18 s/vertex W: 102.81 TB
1,176,178,320,708 r
progress 100.00%
SV6 Extract
1 vertex R: 78.22 MB
4.49 s/vertex W: 78.62 MB
431,928 rows
progress 100.00%
SV4 Extract
1 vertex R: 78.22 MB
5.16 s/vertex W: 78.62 MB
431,928 rows
progress 100.00%
SV1 Extract
1 vertex R: 78.22 MB
4.24 s/vertex W: 49.14 MB
411,210 rows
progress 100.00%
SV10 Cross
209,396 vertices R: 120.03 TB
5.60 s/vertex W: 69.77 MB
382,878 rows
progress 100.00%
SV5 Extract Cross
209,396 vertices R: 2287.11 TB
3.10 min/vertex W: 51.99 MB
421,070 rows
progress 100.00%
SV2 IndexedPartition
1 vertex R: 49.14 MB
8.89 s/vertex W: 50.33 MB
411,210 rows
progress 100.00%
SV3 Process
1 vertex R: 49.14 MB
4.14 s/vertex W: 47.00 bytes
3 rows
progress 100.00%
SV11 Combine
209,396 vertices R: 146.70 MB
0.43 s/vertex W: 51.50 MB
426,337 rows
progress 100.00%
SV12 Process
209,396 vertices R: 51.50 MB
0.44 s/vertex W: 51.50 MB
426,337 rows
progress 100.00%
SV21 Process
209,396 vertices R: 51.49 MB
0.86 s/vertex W: 1.60 MB
95,449 rows
progress 100.00%
SV13 PodAggregate IndexedPa...
838 vertices R: 51.50 MB
6.12 s/vertex W: 59.07 MB
426,337 rows
progress 100.00%
SV22 PodAggregate IndexedPa...
838 vertices R: 1.60 MB
7.29 s/vertex W: 6.88 MB
95,449 rows
progress 100.00%

SV10: Stages are pipelines (sort of)



Vertex Runtime Statistics

Data Read Cross Prod	0.00 bytes
Data Read Intra Prod	0.00 bytes
Average Vertex Execution Time	6s
Total Compute Time	325h 34m 56s
Total Rows Written	382,878
Total Vertex Data Read	120.03 TB
Total Vertex Data to Read	120.06 TB
Total Vertex Data Written	69.77 MB
Vertex Read Failure	0
Vertex Duplicate Discards	186
Vertex Revocations	0
Vertex Scheduled Executions	0
Min Vertex Data Read	78.62 MB
Average Vertex Data Read	601.06 MB
Max Vertex Data Read	40.82 GB
Total Wait Time	21m 12s
Inclusive Time	215h 45m 44s
Exclusive Time	8m 47s
Total Throttling Time	N/A
Operator Wait on IO Time	21m 12s
Max Time in GC	0
Overall Memory Peak Size	419,364,864
IO Memory Peak Size	58,720,256
Managed Memory Peak Size	N/A
Max Row Data Size	2,145
Max Row Count	3,831
Input Total Wait Time	16h 45m 29s
Output Total Wait Time	21m 12s
Input Total Throttling Time	N/A
Output Total Throttling Time	N/A
Input Operator Wait on IO Time	16h 45m 29s
Output Operator Wait on IO Time	21m 12s

How do you express DAGs in Scope?

```
// MULTIPLE CONSUMERS OF @b
@a = SSTREAM “2026/01/01/00/clicks_us_hourly.ss”;
@b = PROCESS @a USING ClickFraudDetector();
@c = SELECT * FROM @b WHERE @b.fraud == true;
OUTPUT @c TO …;
@d = SELECT * FROM @b WHERE @b.fraud != true;
OUTPUT @d TO …;
```

My plan for today

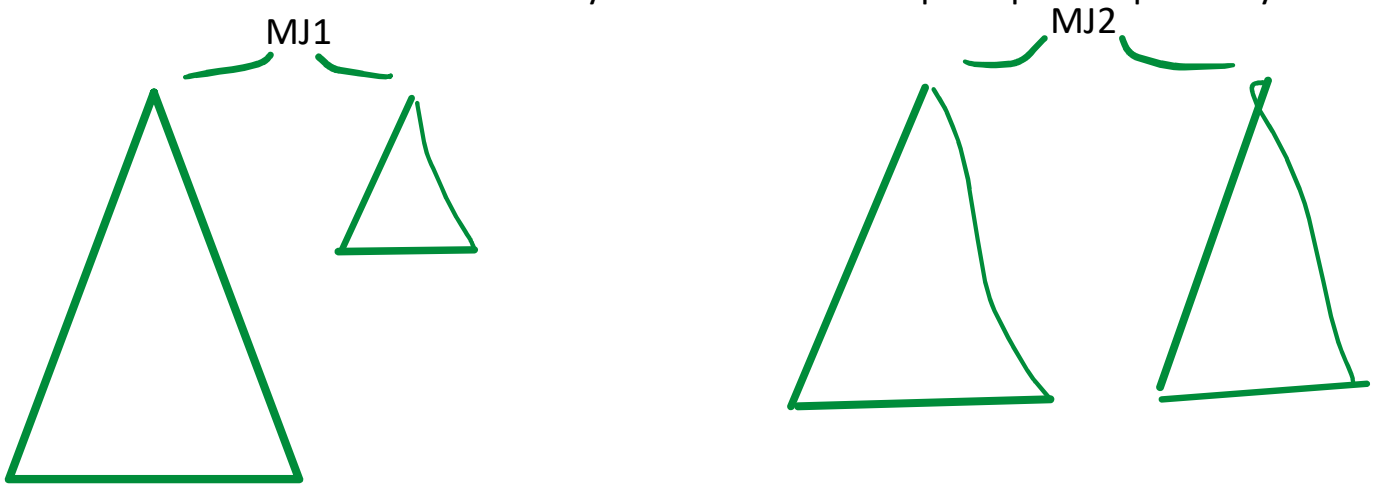
0. Background: Scope (done!)
1. Optimality and the real problems with Cascades
2. Partition (distribution) property
3. DAGs
    - Physical spools
    - Sharing

Story 1: Principle of optimality

- Principle: "A subplan of an optimal plan is optimal."
  - This is the assumption underlying DP & memoization of QO.
- Last week's slide showed Volcano pruning because
  - Cost(Subtree2) = Cost(Node1(Subtree2)) - LocalCost(Node1)
  - This relies on a much stronger assumption: total work metric.
  - i.e. Cost = LocalCost + Subtree costs.
  - (Aside: Not sure how to apply that to branching ops like join.)
- If you use a latency metric…
  - Cost(MJ1(A,B)) = LocalCost(MJ1) + MAX(Cost(A),Cost(B))
  - The Volcano trick is no longer theoretically sound

Metrics and optimality

- But it's much worse than that: latency metrics violate the principle of optimality too!



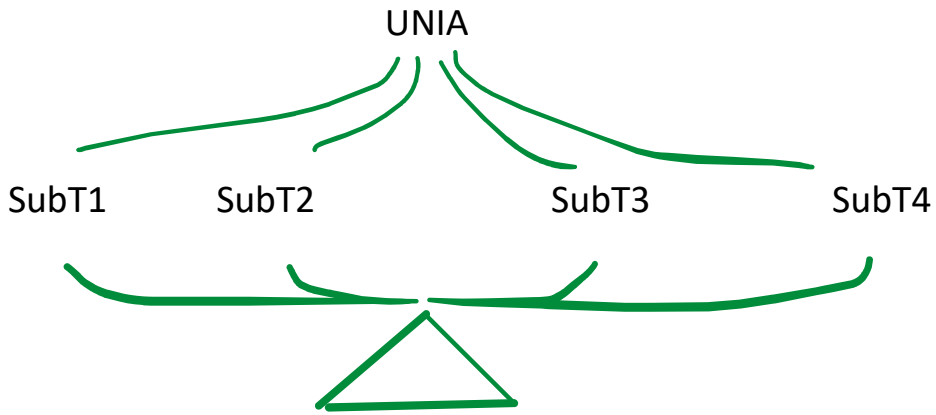
- Cost(MJ1) = 1+ MAX(10,3)
- Cost(MJ2) = 1 + MAX(8,8)
- … but optimizing for distributed or parallel execution requires a latency term

Everything violates the principle of optimality

Any resource constraint violates it.

Pipelining violates it. (Adds a negative term.)

Shared subtrees violate it:



Which subtrees should be charged for S? There's no right answer.

Why do distributed and parallel optimizers even work at all? Via a LEAP OF FAITH.

… but it works well enough

- Optimizers are economically significant
- “Just work” most of the time
- [In Scope] the bigger problem is: customers want different metrics
  - Complex relationship between parallelism and multiple concurrent jobs
  - Some have a deadline, and are cost-insensitive
  - Some just want 99.9% reliability
  - Some grow the number of jobs each day
  - Some grow the size of input to a massive job each day

Story 2: partition property

- Physical plans contain shuffle operations
  - There is no corresponding logical operator
  - Similar to sorting
- In every subplan PhysOp1(T1, … Tn)
  - PhysOp1 requires property Pi of its input Ti
  - Ti must deliver Pi' such that Pi' satisfies Pi

- First let's look at sorting…

Sort property of MJ

SELECT MAX(price) FROM order AS O JOIN orderdetails AS D ON orderid
GROUP BY orderid;

=> $GB_{4.0}(JOIN_{3.0}(O_{1.0},D_{2.0}))$

You implemented $GB_{4.0}$ as $GroupedGB_{4.1}$

(Expects input grouped by value of ordered. Emits a row when orderid changes.)

You are implementing Group 3. What is the SORT reqd by $GroupedGB_{4.1}$?

What about the children?

Cascades note: Implementing $JOIN_{3.0}$ means running all the implementation rules that match $JOIN_{3.0}$.

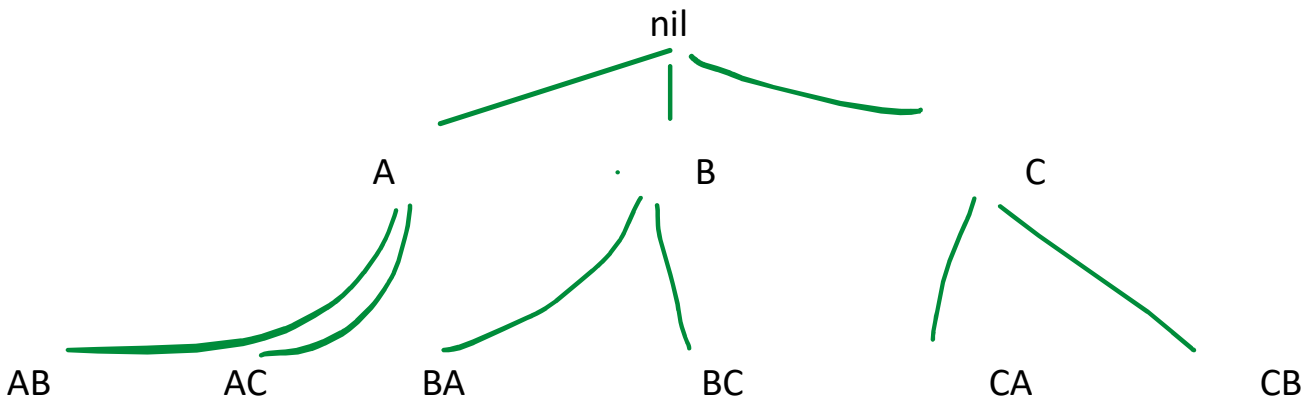
JOIN=>MJ creates a MJ.

JOIN=> HJ creates a HJ.

SORT enforcer=> creates a SORT whose child is group 3 and whose sort reqd = {}.

The sort lattice

- Assume ascending
- “satisfies” relation opposite of “prefix of” relation
  - Sort(A,B,C) satisfies Sort(A,B) satisfies Sort(A) satisfies Sort(nil)



So if reqd = A, you could deliver anything that starts with A.

Physical properties of joins

- In Cascades all decisions are local. Consider 2 join column case:

```
SELECT AVG(price) FROM order AS O JOIN orderdetails AS D
ON O.orderid == D.orderid AND O.date=D.date;
```

You are implementing the JOIN with reqd sort {}.

If you use MJ, what sort do you choose?

You have ? options: any sequence in any asc/desc from {orderid, date}

But you have to decide now!

Partitioning lattice is not finite because of DOP

- P (Type, Columns, DOP)
  - TYPE is HASH or RANGE
  - HASH uses **the** built-in hash function
    - Columns is a bag if hash function is symmetric, sequence otherwise
    - Satisfaction = everything matches exactly
    - P(HASH,[x, x],null) ~~satisfies~~ P(HASH, [x], null)
  - RANGE
    - Columns is a sequence of ordered (asc/desc)
    - [x, x] simplified to [x]
    - Satisfaction is opposite from sort: [x] satisfies [x, y]
    - RANGE case needs a 4th field, the **range spec**, specifying the split points, e.g.
      - {[-inf,7), [7,31),[31,inf]}
  - All fields may be unspecified: P(nil, null, null)

GB(orderid) requires what partitioning?

Partition property for join

- You are implementing Join(A,B) for required partitioning P(nil, null, null). You want to use paired (shuffled, in spark) merge join.
- What required partitioning do you assign A and B? What DOP?
- You can pick anything. But they have to be the same!
- You must assign MJ1 a ground partitioning right now.
- If you want multiple options for DOP, you have to create multiple MJs now.
- Any JOIN or UNIA must do this.

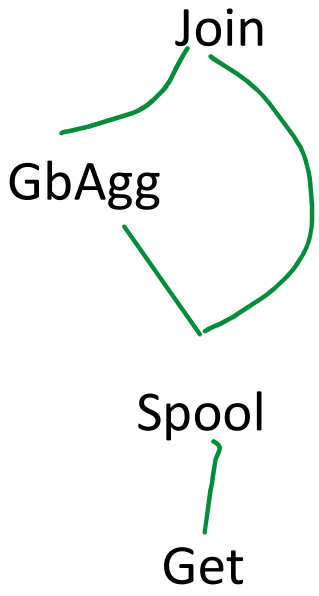
Story 3: DAGs

Spool operator

Scope and SQL Server insert an operator to represent multiconsumer rowsets.

SELECT *, SUM(price) OVER (PARTITION BY orderid) FROM T;

```
Join on orderid     4
| GbAgg on orderid  3
| | Spool Sp1       2
| | | Get T         1
| Spool Sp1         2
```



Shared subtrees will be shared

- Opposite decision made in SQL Server.
  - Why might it be better to expand? To share?
- Problem: How to do implementation?
- By construction, spools end up in their own groups.
- Group 2 will be optimized for each consumer, potentially with different sort and partitioning reqd.
- We do not allow enforcers in spool groups.

Spools: after implementation they get unshared

```
Join on orderid     4
| GbAgg on orderid  3
| | Spool Sp1       2
| | | Get T         1
| Spool Sp1         2

MJ                      4.1
| GroupedGbAgg              3.1
| | PhySpool                2.1
| | | Clustered index scan T 1.1
| PhySpool                  2.2
| Clustered index scan T    1.2
```

Reshare them again (credit to Nico Bruno)

After optimization, restitch them back together with any fixup (enforcers)
Mini-optimization problem: Choose the cheapest such plan.

MJ 4.1
| GroupedGbAgg 3.1
| | PhySpool 2.1
| | | Clustered index scan T 1.1
~~| PhySpool 2.2~~
~~| Clustered index scan T 1.2~~
| // You might need enforcers here
| PhySpool 2.1