Vectorized Interpreters

MRT for PL

Spring 2023

Greetings!

- I'm a PL person who's worked on a bunch of compilers and also done some data-systems work.
- Industrial programmer, not academic. No grad school.
- Lindsey asked me to give a talk to Languages, Systems and Data (LSD) group.
- Said "just about any topic" and this is what's been on my mind lately.

Overview

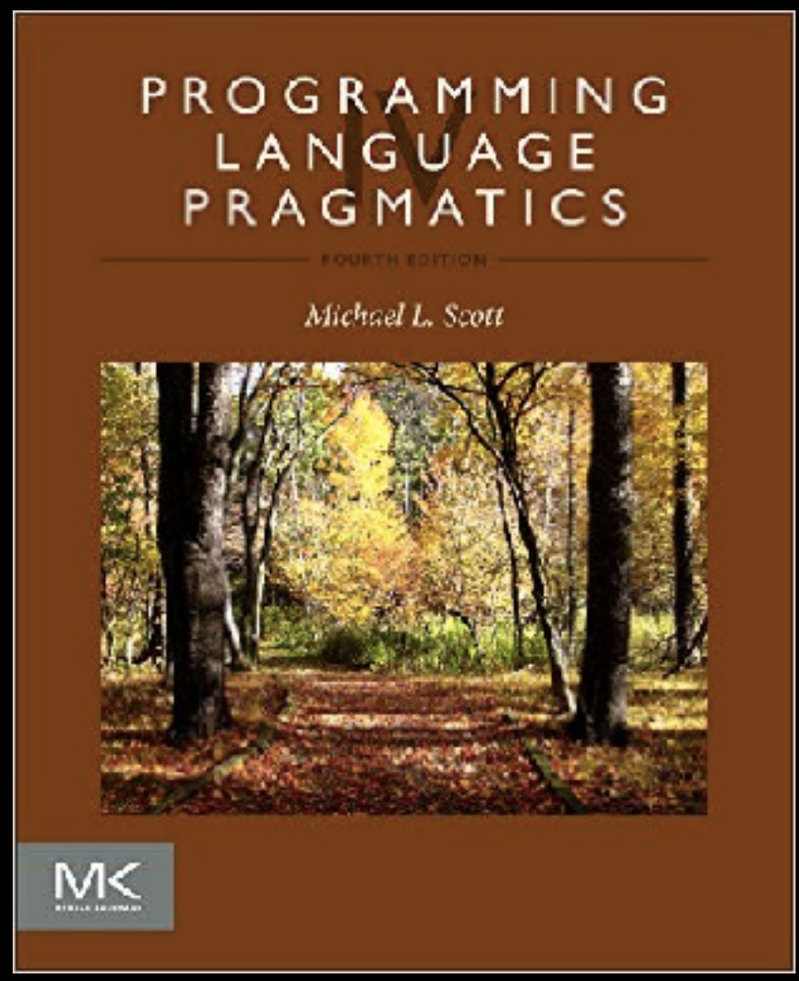
- Topic: One Special Technique in **PL pragmatics**
    - Motivation & analogy for **vectorized interpreters**
    - Explain how technique works, what you get
    - Talk about which languages employ it
    - Talk about details, caveats, opportunities

Part 1: Introduction

Why am I even talking about interpreters?

PL Pragmatics

- Implementation strategies and mechanisms:
  "How $FEATURE actually works"
  - Compilation, linking, versioning
  - Interpretation, introspection, sandboxing
  - Binding, dispatch, inlining, unwinding
  - Control flow, evaluation order, iteration
  - Scope, lifetimes, GC, environments
  - Data representation, layout, indirections
  - Parallelism, barriers, atomicity, locking
  - Etc. Etc. Etc. Etc.
- If I understand correctly (certainly judging from texts) this isn't taught very deeply. Seems like mostly practical lore.



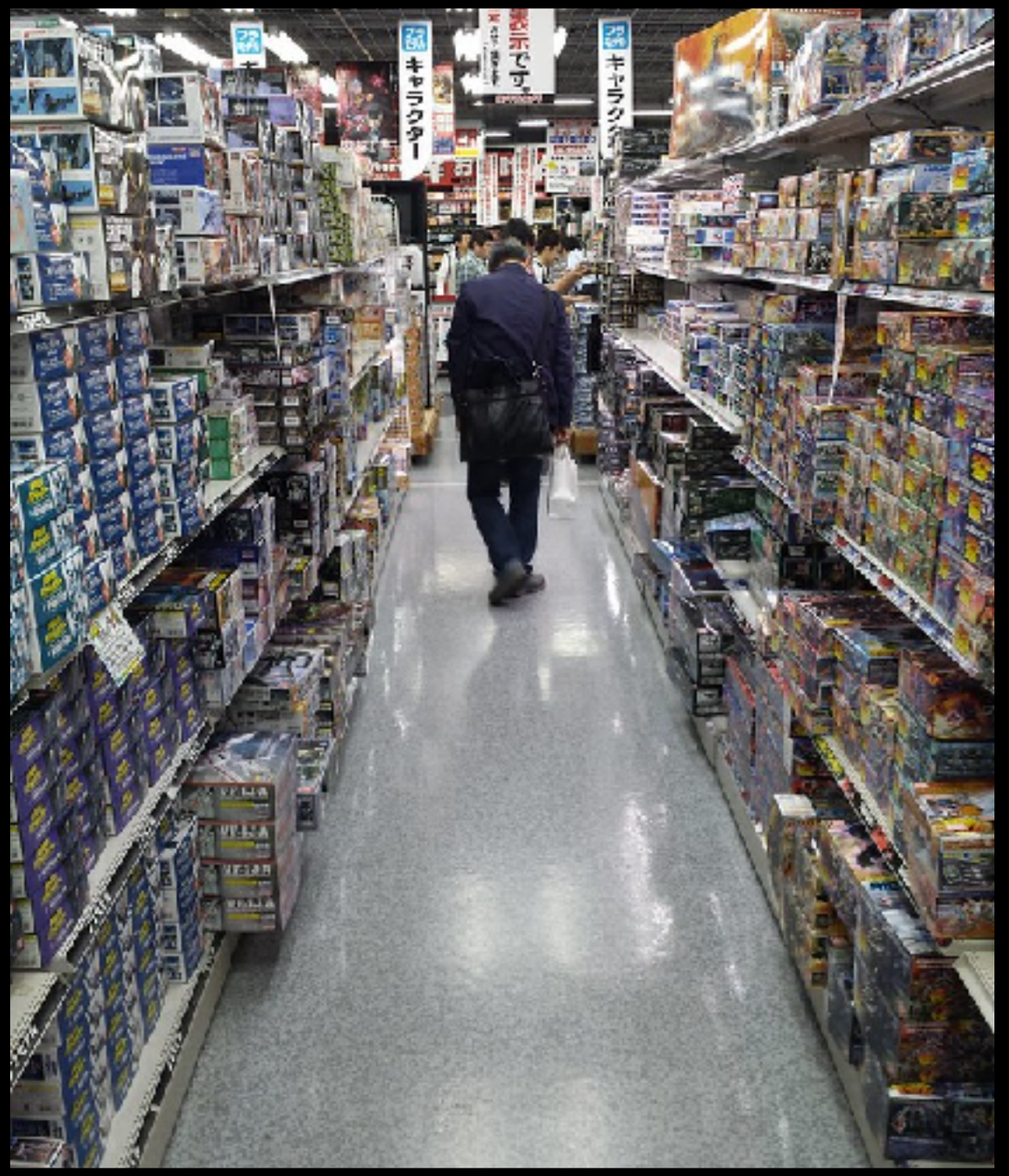
maybe the only book on the subject?

Pragmatics matter!

- Users complain constantly that compilation takes too long.
- Sadly, also complain constantly that interpreters are too slow.
- Field of compilers actually originates here. First compiler (A-0) motivated by making interpreter dispatch faster, pre-processing all dispatch choices once for a given program.
- A.K.A. 1st Futamura projection.

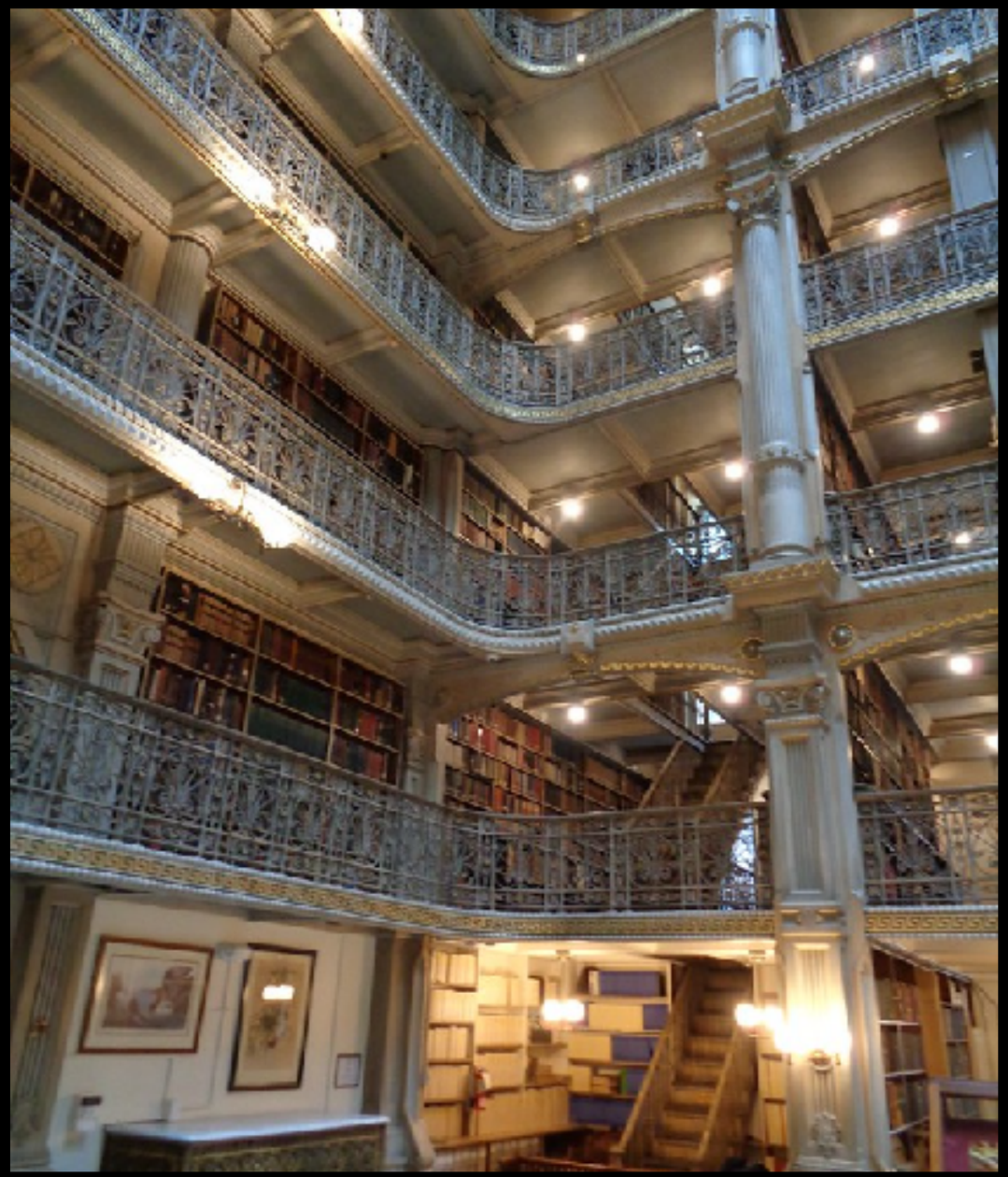
Not just speed

- Tons of pragmatic metrics
  - Footprint & update size
  - Fault tolerance
  - Easy resource management
  - Easy debugging
  - Field extensibility
    (late binding, reflection)
- Users care a lot about all of it!



Not just users

- Implementors have a laundry list of pragmatic concerns too.
  - Labor cost of implementing
  - Labor cost of other tooling (REPL, debugger, profiler)
  - Ease of safety engineering
  - Language evolvability
  - ... (lots of others) ...
- Many serious concerns!



Compilers: not so great!

	Interpreters	Compilers
edit-test cycle latency	✅	❌
program / update size	✅	❌
field extensibility	✅	❌
execution speed	❌	✅
total system throughput	❌	✅
labor cost of implementing	✅	❌
labor cost of tooling	✅	❌
ease of safety engineering	✅	❌
language evolvability	✅	❌
... (lots of others) ...	✅	❌

Beginning of an Analogy

- Broadly speaking: interpreters are better than compilers at almost everything pragmatic that users and implementors care about.

- Except execution speed (and throughput) which we sacrifice everything else for, choosing compilers.

- Where else have we seen this sort of tradeoff being made?

Driving: not so great!

	Walking	Driving
short trip latency	✅	❌
urban space consumption	✅	❌
broad accessibility	✅	❌
speed once underway	❌	✅
total system throughput	❌	✅
cost of building infrastructure	✅	❌
cost of power and maintenance	✅	❌
ease of safety engineering	✅	❌
power-source flexibility	✅	❌
... (lots of others) ...	✅	❌

Is there another option?

(besides bikes, which complicate the analogy)

19th Century Tech Calls

- MRT: mass rapid transit!
- Full industrial-powered speed
- Shared Vehicle costs amortized over many simultaneous riders
- Density and uniformity gives even higher total throughput!
- Hope analogy not too strained
- Let us briefly consider it

MRT: compelling!

	Walking	Driving	Subways
short trip latency	✅	❌	✅
urban space consumption	✅	❌	✅
broad accessibility	✅	❌	✅
speed once underway	❌	✅	✅
total system throughput	❌	✅	✅✅✅
cost of building infrastructure	✅	❌	OK
cost of power and maintenance	✅	❌	OK
ease of safety engineering	✅	❌	✅
power-source flexibility	✅	❌	✅
... (lots of others) ...	✅	❌	✅

Vectorized Interpreters: similarly compelling!

	Interpreters	Compilers	Vectorized Interpreters
edit-test cycle latency	✅	❌	✅
program / update size	✅	❌	✅
field extensibility	✅	❌	✅
execution speed	❌	✅	✅
total system throughput	❌	✅	✅✅✅
labor cost of implementing	✅	❌	🆗
labor cost of tooling	✅	❌	🆗
ease of safety engineering	✅	❌	✅
language evolvability	✅	❌	✅
... (lots of others) ...	✅	❌	✅

Tragic Neglect

- Tragically neglected technique in PL literature, education
- Most introductory texts, courses just don't mention it
- Most practitioners: blank stare
- Compare: vast literature on hyper-complex JIT tech
- Compare: electric car tech
- "Maybe just build more MRT"?

Part 2: Technique

How does this actually work?
(And what do you get out of it?)

Two prongs

- Data will mainly be stored in vectors of primitives
    - Data structures get decomposed to vectors ("columns")
- Control will mainly involve applying vector operators
    - Control structures get decomposed to operators *or* data

Data decomposition

```
garden: plants[N]
```

```
struct plant {
  color: enum,
  flower: bool,
  height: f64
}
```

`garden[0]` `garden[1]` `garden[N-1]`

Decomposed to vectors:

`colors` `flowers` `heights`

Data decomposition

```
menagerie: critter[N]
```

```
enum critter {
  opossum(dead:bool)
  snake(length:f64)
  owl
}
```

`menagerie[0]` `menagerie[1]` `menagerie[N-1]`

Decomposed to vectors:

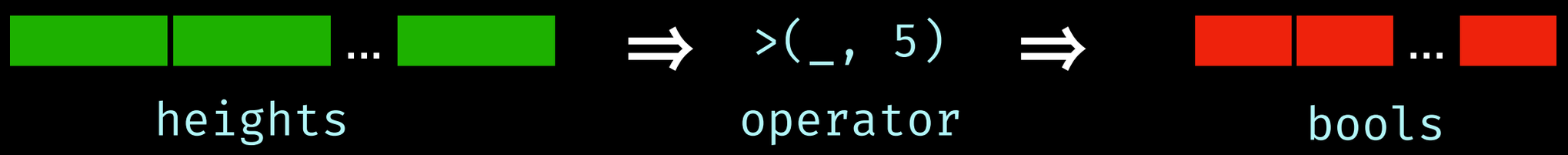
`case tag` `opossums` `snakes`

Control decomposition

```
garden: plants[N]
```

```
for p in garden {
    if p.height > 5 {
        ...
    }
}
```

Decomposed to vectors:



Two complementary prongs that deeply refactor language

- Vectorized data enables/requires vectorized operators
- Vectorized control uses *both*, refactors language:
  - Loops still happen but no longer in user code, all loops implicit in calls to vectorized operators
  - Conditionals fundamentally change nature, turning from control into vectorized data (`bool` vectors)

Two complementary prongs that deeply refactor language

- No more of this:

```
for i in iota(1,N) { if foo(i) { x += i } }
```

- Users now write something like:

```
iota N | fork | (_, foo) | select | sum
```

- Or if you prefer:

```
select sum(i) from iota(1,N) as i where foo(i)
```

- Expect your PL to change. You can hide a lot of this in sugar if you want to "look normal", but hiding it completely is "solving auto-vectorization".

Control change is fundamental

- What is really happening here?
  - User control factored into <u>inner loops</u> and <u>outer paths</u>
  - System provides <u>fixed repertoire</u> of inner loops
  - Analogy: fixed rail lines, individual trip routes
  - Arguable (PL philosophy): "imperative" → "declarative"
  - Arguable (poli-sci): "laissez-faire" → "central planning"

Why does this matter?

- If user control is only outer paths, it doesn't matter if those paths are chosen in compiled or interpreted code
- Interpreter opcode-dispatch loop is only an outer loop
- One unpredictable indirect dispatch per thousand data elements is lost in the noise: interpreter cost is amortized

Best-of-both pragmatics

- Infinite number of user programs
  - Choose interpreter pragmatics for outer paths
- Finite number of vector operators shared by all programs
  - Choose compiler pragmatics for inner loops

Remember this table?

	Interpreters	Compilers	Vectorized Interpreters
edit-test cycle latency	✅	❌	✅
program / update size	✅	❌	✅
field extensibility	✅	❌	✅
execution speed	❌	✅	✅
total system throughput	❌	✅	✅✅✅
labor cost of implementing	✅	❌	🆗
labor cost of tooling	✅	❌	🆗
ease of safety engineering	✅	❌	✅
language evolvability	✅	❌	✅
... (lots of others) ...	✅	❌	✅

Ok but what's that about "total system throughput"?

✅✅✅

Throughput freebies

- Amortizes pointer chasing across many data elements
- Avoids cache pressure wasted on cold columns
- Can be denser for sparse data (unions, optionals)
- Avoids padding for data alignment
- Avoids branch mispredicts while in operator loops
- Often naturally SIMD (including "SIMD" of bitwise ops)
- Often naturally sequential & prefetchable access patterns
- Several new optimization opportunities (will return to this in part 4)

Ok but what's that about "labor cost of implementing and tooling"?

🆗

Partly that "finite number" of inner loops

- Vectors of primitives means only a few vector types
- Plausibly just say: `bool`, `byte`, `i64`, `f64`
- NB: vector of `bool` is bitmap, no more `byte` per `bool`
- (Which is good because there'll be a lot of `bool` vectors)

Partly that "finite number" of inner loops

- Few types is good, because interpreter will contain one precompiled specialization of each operation per type
  - Actually more like `(T*2)^A`
    - `T` is number of types
    - `2` is because vector vs. scalar
    - `A` is arity of the operation, usually only 1-3
- Likely hundreds of specialized operators (lots of macros!)

Also that "decomposition" of data and control

- User mental model can be bent a fair bit, but..
  - Many users *like* to think about `struct` and `union`
  - Many users *like* to think about `if`, `each` and `while`
- Decomposition is imperfect, visible, confusing: abstraction leaks
  - "Why can't I just write a loop instead of maps and filters?"
  - "My function is slow, can you add a new vector operator?"
- PL design and tooling carries some extra costs from such leaks

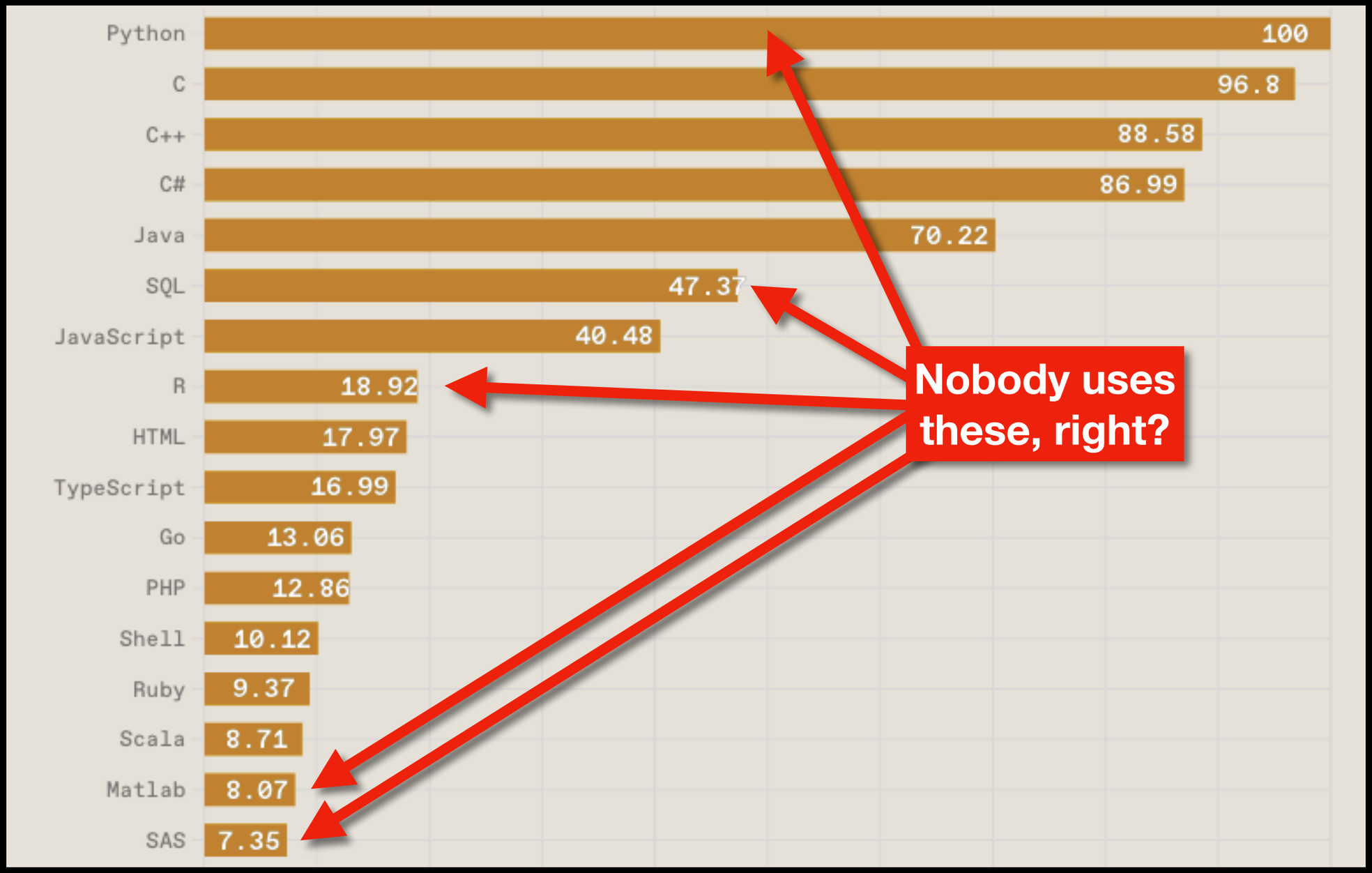
A return to an analogy

- Life is all compromises, in PL as in urban transport:
  - "Why won't the train arrive already, I'm late!"
  - "Why must I listen to this stranger's phone call?"
  - "My commute is long, can you add a new line?"
- My opinion: it's worth knowing the MRT option *exists*!

Part 3: Examples

Which languages do this?
(And do any of them have users?)

IEEE Spectrum's Top Programming Languages 2022

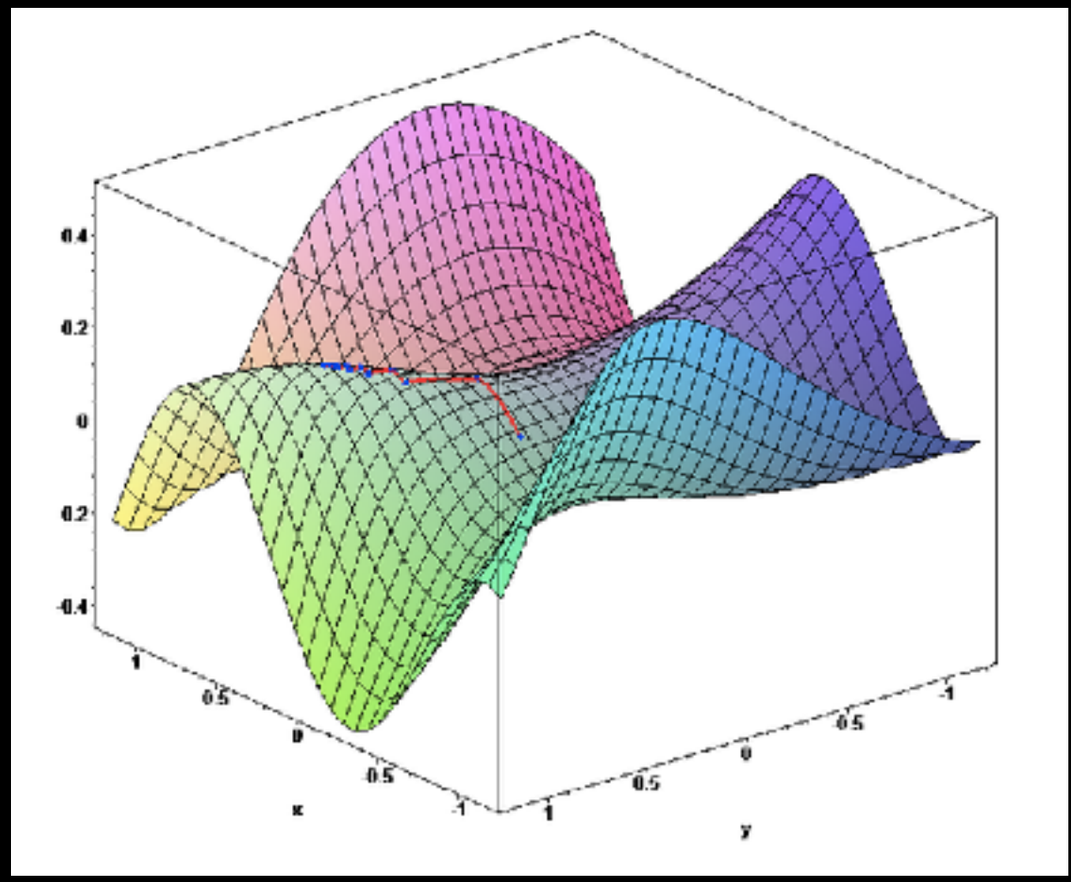


They're Kinda A Big Deal

- Several of the most widely used languages do this
- They are languages that PL people tend to overlook
- Lots of "idiosyncrasies" at best, or serious design flaws
- Not going to defend any of these as PLs, not the point
- Point is: vectorized interpreter technique works ok

Aside: Python?

- Yes
- Python tops the charts because ML and Data Science
- These use PyTorch and NumPy respectively
- Both fine examples of vectorized interpretation



Aside: SQL?

- Yes
- With a caveat: SQL DBs split into OLTP and OLAP
- Transaction processing (point access) vs. bulk analytics
- OLTP DBs typically store data in rows, OLAP in columns
- This line is blurring, for reasons we'll touch on it in part 4

Common themes

- Many similarities in popular languages that do this
  - Numerical analysis, data analysis, data transfer
  - Obvious choice if data and ops already homogeneous
  - Not as obvious with heterogeneous data, but see convergent evolution in many data-intensive systems

```
foreach <many-data>: do <same-thing>
```

Another perspective

- Ever heard of "Ousterhout's Dichotomy"?
  - Fast, static, compiled language on the inside
  - Slow, dynamic, interpreted glue language outside
- A lot of these are "glue language for Fortran code"

Everything is a sandwich

- Arguably PHP, Perl, AWK, sed
- Vector: string
- Operators: regex match, string interpolation, IO
- Short programs applied to lots of data
- My old boss: "The fast Lisp is the one that spends most of its time in its primitive C functions"

Everything is a sandwich

- Arguably also shell, though only in the lives of the subprocesses that it runs
- Vector: `char[PIPE_BUF]`
- Operators: `/usr/bin`
- Composition via pipes, shell loops barely work
- Those fun stories about shell scripts beating other PLs are real though, because the "operators" are fast!

Have you heard?

- I have to mention APL eventually
- APLs are <u>not</u> especially popular (outside the cult)
  - Tacit, mathy, higher-order, wild extended charset
  - Every program a code-golf hole in one
  - Example: `((+.×⍤⊢⍤⍨∘.×⍤⍨)1↓⍳)17`
- But they <u>are</u> very fast, tiny vectorized interpreters

Other cases #1

- "Data oriented design"
  - Peak-perf-chasing trend in game dev
  - Sometimes an interpreter (scripted "game engine")
  - Sometimes an "SoA feature" in a compiled PL: Jai
  - Sometimes just a "design pattern": C++, Zig, Rust, often with library / metaprogram / macro support

Other cases #2

- "Nested Data Parallel" languages
  - NESL, Nepal, Manticore, Futhark, Legion, ...
  - Some interpreters, some compilers
  - Mainly aiming to extract parallelism from vectorized code. Also do a "flattening transformation" on nested vectors: https://en.wikipedia.org/wiki/Flattening_transformation (One of several extensions to data structure decomposition)
- Today's topic (mainly) much simpler: amortizing interpreters

Reiterating first slide

- This is quite a widespread pattern
- Often in somewhat overlooked PLs
- Nonetheless many are load-bearing for real work
- Again: the technique is worth studying and reusing
- Amortized interpreters via homogeneous data & ops

homogeneity = efficiency

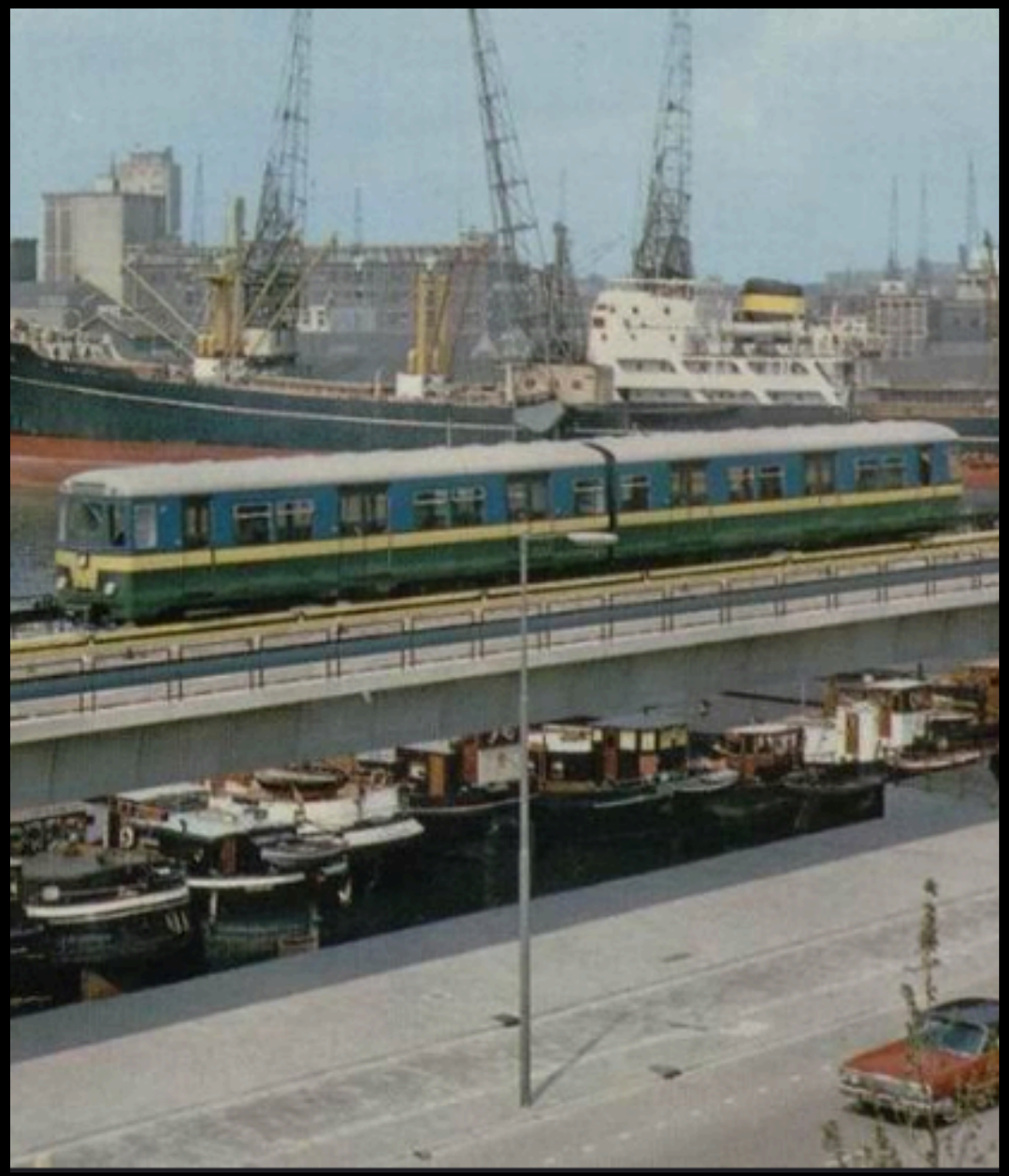
(via cost amortization)

Part 4: Details

Some caveats and opportunities

Moderate chunk size

- Operating on an whole vector at a time has some drawbacks
- Better to use a vector chunk
- Chunk size is minimum unit of work, too big can mean waste
- Big chunks use more cache, memory bandwidth
- (Arbitrarily long trains also somewhat unwieldy!)



Moderate chunk size

- This is a quantitative question
- It has been investigated and answered fairly conclusively
- Your chunks should be 1-8kb
- I.e. amortizing interpreter costs >1000:1 is enough for those costs to be ~ignored
- NB: L1 dcache usually 32kb due to 8-way VIPT, 4k pages

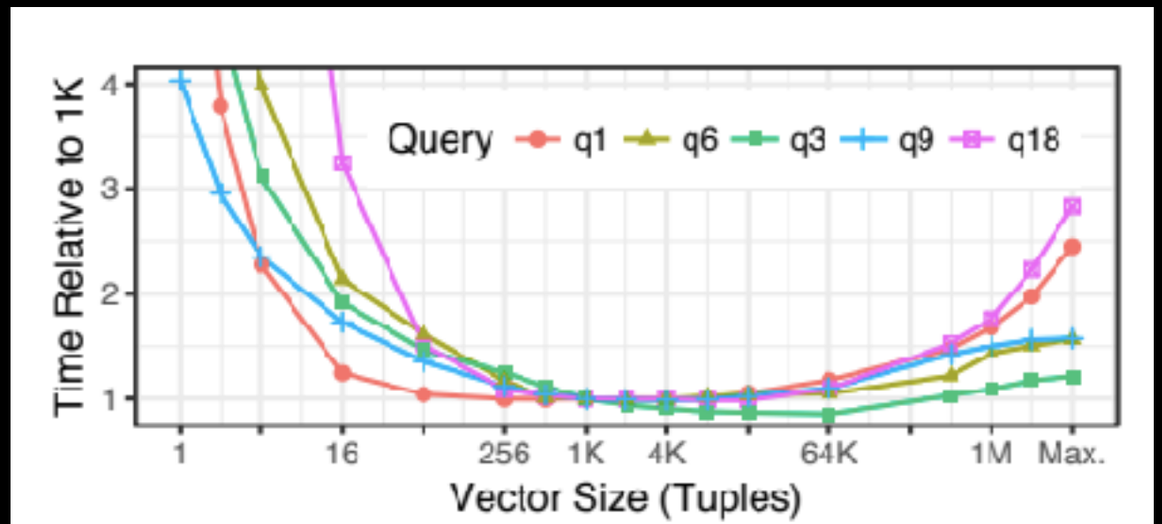


Figure 5: **Tectorwise Vector Sizes** – *Times are normalized by 1K vector time.*

https://doi.org/10.14778/3275366.3275370
Kersten et al. 2018

PAX / HTAP / etc.

- Workload might want to do a pointwise load/store on a row, access 1 whole row
- Large columns are bad for this! Need to read and write lots of wasted data
- Another reason to use partial columns: can store all columns for a group of N rows
- Bunch of literature jargon:
  - PAX ("Partiton Attributes acXross")
  - HTAP (Hybrid Transactional / Analytic)
  - "Row Groups", "Data Blocks", etc.
- Often mixed with LSM design: reform row group to columns when flushing to disk

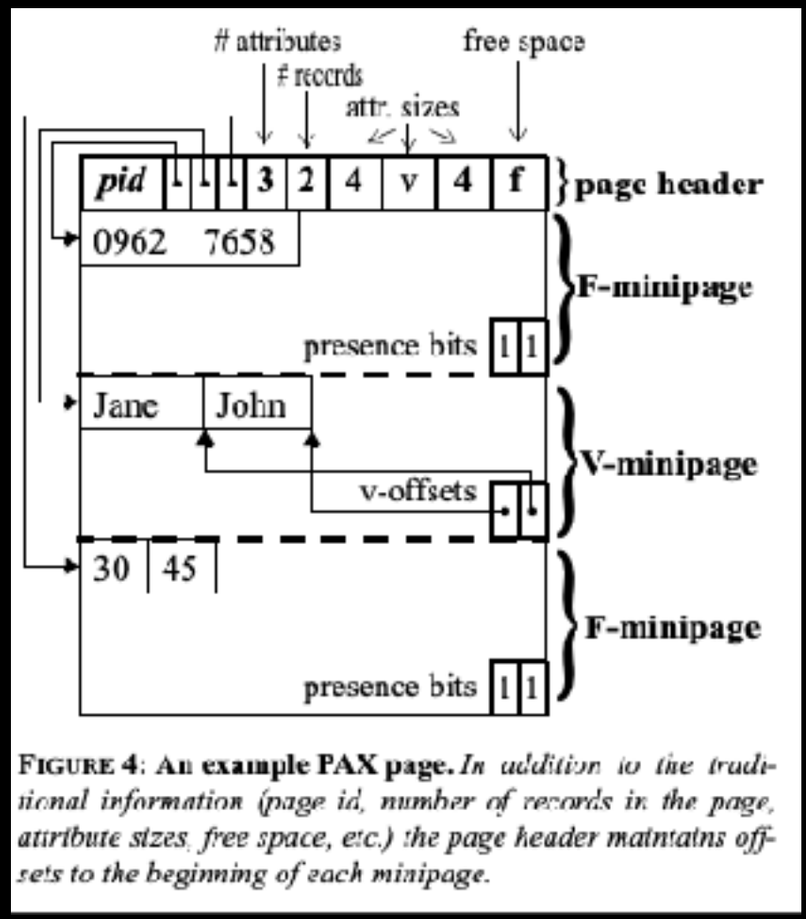


FIGURE 4: **An example PAX page.** *In addition to the traditional information (page id, number of records in the page, attribute sizes, free space, etc.) the page header maintains offsets to the beginning of each minipage.*

https://doi.org/10.1007/s00778-002-0074-9
Ailamaki et al. 2002

Parallelism

- As mentioned, vector ⇒ SIMD
- People have been trying to extract parallelism through vectorization forever
- TI ASC, CDC STAR, Cray, Convex, Thinking Machines
- Eventually we all got GPUs and Arm SVE2
- GPU analytic databases and vectorized interpreters are definitely a thing now (OmniSci, RAPIDS, ArrayFire)

Distribution

- Many 2023 OLAP databases aren't single-node
- "Big Data" system == distributed vectorized interpreter
  - Amortizing network RPCs, not just indirect branches!
  - Parallelizing as much as possible too!
- Parquet, ORC, Arrow. About 8 million Apache projects.
- Redshift, BigQuery, Snowflake ...

Compression

- Data stored in a column is often very similar row-by-row
- You can often compress it!
- Don't laugh: "lightweight" only
  - RLE, delta encoding, null skipping, constant runs, etc.
  - Many operators can act directly on compressed data, in situ
  - Even decompressing then acting can be a memory bandwidth win, decompress values only in cache

Declarative reordering

- Earlier I mentioned that vectorization makes program control more "declarative"
- Formally: less dependent on some stated evaluation order
- This is an opportunity to optimize execution!
- SQL join order can vary speed by >4 orders of magnitude!

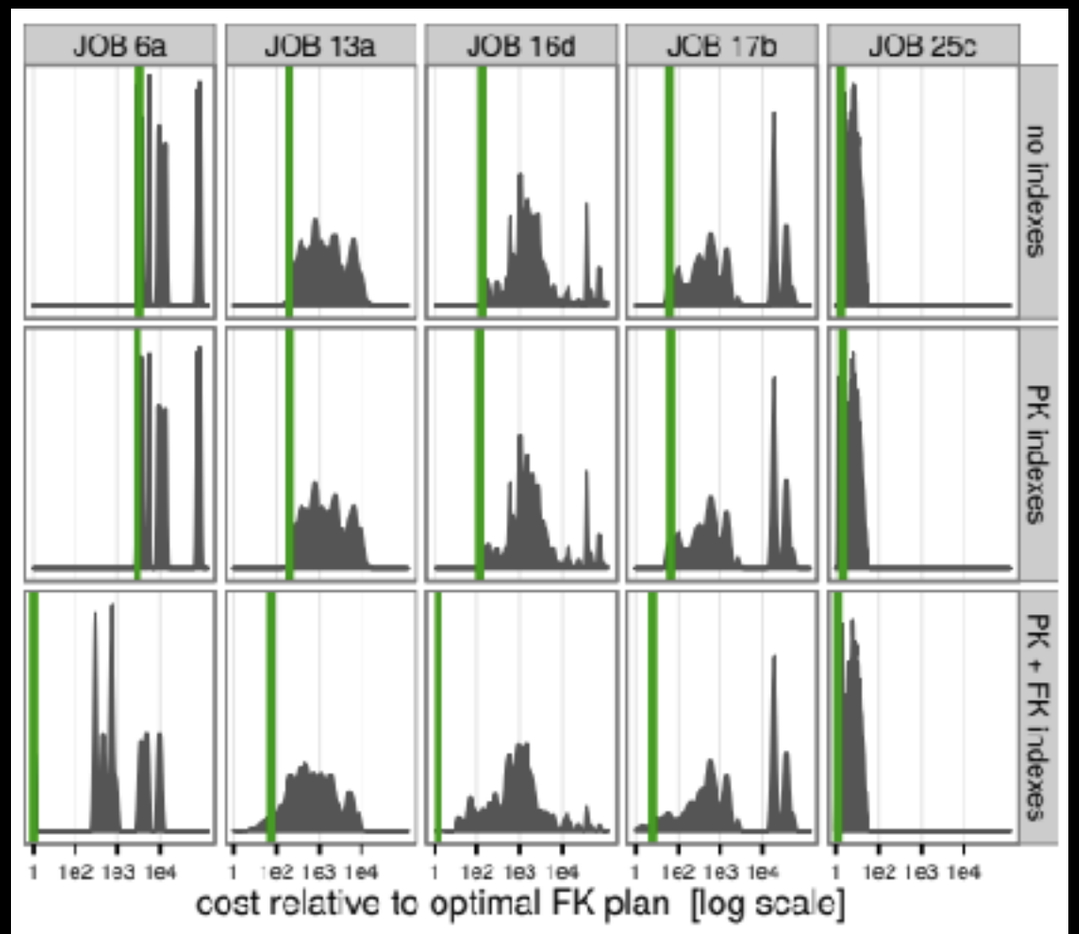


Figure 9: Cost distributions for 5 queries and different index configurations. The vertical green lines represent the cost of the optimal plan

https://doi.org/10.14778/2850583.2850594
Leis et al. 2015

Adaptive reordering

- Great thing about interpreters: can change stuff at runtime
- Join order can adapt on the fly
- Get part way in, change plan
- If you have CPU to spend, try two at once and cancel the one that's going too slow

Conclusion

Thank you for entertaining this
extended mass rapid transit analogy

Could go on all day

- Database literature is full of ideas that PL people should read and think about!
- Array languages have a bad reputation for being all math!
- Everything should be vectors!
- Interpreters can be amortized!
- The memory hierarchy is real!
- Invest in mass rapid transit!
- I am out of time

Fini