Abst… Speed Up A Pyth… …,000x

The views and opinions expressed herein are mine and like totally do not necessarily represent or reflect those of The Walt Disney Company or any other entity, not even my mother.

Know the stack fro… …eering up
through your custome… …nd obsess
about it during time you w… …a social
life. Tonight, you'll learn how o… …p a
critical program by 114,000 times, s… …e a speed
guru too. As to your social life, you're at … …g, aren't
you? Hmmm.

davidschachter@gmail.com

Abstract: How to Speed Up A Python Program by 114,000x

Know the stack from raw silica and computer engineering up through your customers' needs, code for 40 years, and obsess about it during time you would otherwise waste on a social life. Tonight, you'll learn how one Pythoneer sped up a critical program by 114,000 times, so you can become a speed guru too. As to your social life, you're at this meeting, aren't you? Hmmm.

Bio Addendum

In his spare time, Mr. Schachter serves as a standard unit of tungsten for the National Bureau of Standards. With his husband, Mr. Schachter lives in Northern California. Without his husband, he lives in a van down by the river.

MuffinMavens™ Presentation Checklist

- ❑ Start with a joke
- ❑ Imagine the audience in your underwear
- ❑ Use "Comic Sans" for the slides

And now our Feature Presentation

Presentation Structure

1. Problem: Running Many Jobs Took Too Long
2. The Algorithm
3. The Solution in 2011: Code Mechanix
4. The Solution in 2012: Computer Architecture
5. Future Steps
6. Part II: Lessons Learned

Report Tool

For each of hundreds of questions, evaluate 500 different ways to analyze the data by taking random samples of that data. The data for each question is large, up to hundreds of millions of rows.

Feb. 2011 Prototype

Gave accurate results but took 19 hours for a subset of the processing, conflicting with 4.5 hour batch window.

Increasing Data Volume

The Algorithm

Some original rows will be used more than once. Some rows will not be used at all.

```
jobs = cursor.execute(JOBS_SQL)
for job in jobs: # Outer loop: hundreds of jobs
  rows = cursor.execute(DATA_SQL % job)
  data = numpy.array(rows)
  for i in range(500): # Middle loop: 500 samples
    resample_indices = random_array(len(data))
    # Three inner loops: millions of rows
    new_data = data[resample_indices]

    sums_T[i] = numpy.sum(new_data[where  something])
    sums_F[i] = numpy.sum(new_data[where !something])

  cursor.insert(generate_sql(sums_T))
  cursor.insert(generate_sql(sums_F))
```

Sum down each column.

The Algorithm

```
jobs = cursor.execute(JOBS_SQL)
for job in jobs: # Outer loop: hundreds of j
  rows = cursor.execute(DATA_SQL % job)
  data = numpy.array(rows)
  for i in range(500): # Middle loop: 500 s
    resample_indices = random_array(len(data))
    # Three inner loops: millions of rows
    new_data = data[resample_indices]

    sums_T[i] = numpy.sum(new_data[where  something])
    sums_F[i] = numpy.sum(new_data[where !something])

  cursor.insert(generate_sql(sums_T))
  cursor.insert(generate_sql(sums_F))
```

Wait on RDBMS.

Wait on RDBMS.

Repeated random reads.
Bulk writes.

Sequentially, *twice*, read the data just written.

Wait on RDBMS.

The Speedups

Mechanical: 8 weeks in 2011, 114X faster

1. *Hoist invariant code and precompute values*
2. *Introduce pipeline parallelism*
3. *Use* **`numpy`** *effectively*
4. *Parallelize with* **`multiprocessing`**

Computer architecture: 7 weeks in 2012, 111X faster

Hoist Invariant Code & Precompute

Original code had duplicate facts, magic numbers, poor structure. Two weeks to clean it up, reveal underlying structure, precompute four columns.

- Faster
- More readable

2-5x speedup

```
jobs = cursor.execute(JOBS_SQL)
for job in jobs: # Outer loop: hundreds of jobs
  rows = cursor.execute(DATA_SQL % job)
  data = numpy.array(rows)
  for i in range(500): # Middle loop: 500 samples
    resample_indices = random_array(len(data))
    # Three inner loops: millions of rows
    new_data = data[resample_indices]

    sums_T[i] = numpy.sum(new_data[where  something])
    sums_F[i] = numpy.sum(new_data[where !something])

  cursor.insert(generate_sql(sums_T))
  cursor.insert(generate_sql(sums_F))
```

Introduce Pipeline Parallelism

Split program into Fetcher→Analyzer→Writer

- Faster due to overlap of RDBMS and compute
- Simpler programs
- Quicker re-runs

```
jobs = cursor.execute(JOBS_SQL)
for job in jobs: # Outer loop: hundreds of jobs
  rows = cursor.execute(DATA_SQL % job)
  data = numpy.array(rows)
  for i in range(500): # Middle loop: 500 samples
    resample_indices = random_array(len(data))
    # Three inner loops: millions of rows
    new_data = data[resample_indices]

    sums_T[i] = numpy.sum(new_data[where  something])
    sums_F[i] = numpy.sum(new_data[where !something])

  cursor.insert(generate_sql(sums_T))
  cursor.insert(generate_sql(sums_F))
```

2X speedup, then less over time

Fix numpy For Speed

Convert database result set from string, floats, and ints to all floats (using a lookup table for the strings)

- Replaced PyObject w/ float for full-speed numpy
- Obsoleted in late 2011

8x speedup?

```
jobs = cursor.execute(JOBS_SQL)
for job in jobs: # Outer loop: hundreds of jobs
  rows = cursor.execute(DATA_SQL % job)
  data = numpy.array(rows)
  for i in range(500): # Middle loop: 500 samples
    resample_indices = random_array(len(data))
    # Three inner loops: millions of rows
    new_data = data[resample_indices]

    sums_T[i] = numpy.sum(new_data[where  something])
    sums_F[i] = numpy.sum(new_data[where !something])

  cursor.insert(generate_sql(sums_T))
  cursor.insert(generate_sql(sums_F))
```

Use multiprocessing

Switch from one process to #-of-cores processes (not *n* processes!)

- Use fork() w/o exec()
  ... or ...
- Use shared memory

Near-linear speedup

```
jobs = cursor.execute(JOBS_SQL)
for job in jobs: # Outer loop: hundreds of jobs
  rows = cursor.execute(DATA_SQL % job)
  data = numpy.array(rows)
  for i in range(500): # Middle loop: 500 samples
    resample_indices = random_array(len(data))
    # Three inner loops: millions of rows
    new_data = data[resample_indices]

    sums_T[i] = numpy.sum(new_data[where  something])
    sums_F[i] = numpy.sum(new_data[where !something])

  cursor.insert(generate_sql(sums_T))
  cursor.insert(generate_sql(sums_F))
```

Post-2011 Speedup Analyzer

```
with open("%s/data.bin" % dir, "rb") as handle:
  data = numpy.array(handle.read()))

p = [mp.Process(computeWrapper, data) \
      for i in range(mp.cpu_count())]
[process.start() for process in p] # Middle loop 1
[process.join() for process in p]

def compute_wrapper(data):
  for i in range(500/ mp.cpu_count()): # Middle loop 2
    resample_indices = random_array(len(data))
    # Three inner loops: millions of rows
    new_data = data[resample_indices]

    sums_T[i] = numpy.sum(new_data[where  …])
    sums_F[i] = numpy.sum(new_data[where !…])

  cursor.insert(generate_sql(sums_T))
  cursor.insert(generate_sql(sums_F))
```

Statically scheduled parallelism creates a speed bump at `join()`.

davidschachter@gmail.com

The Speedups

Mechanical: 2011, 8 weeks: 114X

Computer architecture: 7 weeks in 2012, 111X faster

1. *Eliminate copying the big data*
2. *Reduce random number generation*
3. *Touch big data once only by swapping loops*
4. *Use Cython and hand-optimize the C code*

Eliminate Copying the Big Data

Very effective.
A big clue!

1. Reduce row width from 84 to 36 bytes
2. Eliminate all copying of the big data
   - Use radix sort to count how many times each input row should appear in the synthetic data set
   - Gives a “virtual” synthetic data set; no need to actually create it
   - Eliminates random reads
   - Eliminates sequential writes
   - Takes advantage of the unimportance of row order

   Replace:
   ```
   resample_indices =     random_array(len(data))
   ```
   with:
   ```
   bincounts = radix_sort(random_array(len(data)))
   ```
3. Eliminate vector operations from inner loop (the “`where`” clause)

Reduce Time for RNG

Reduced user time exposed system time as the next target. Tracing showed too many calls to /dev/random.

Reduced RNG Linux kernel calls from `500*len(data)` to `500+len(data)`. Here's how:

1. Generate one list of *n* random ints in [0, n) where `n=len(data)`
2. Radix sort the list to count the unique values: a new list
3. Append a copy of that list to itself, making a list of length 2*n*.
4. Generate a list of 500 random ints between 0 and *n*-1.
5. Use the second list to index into the first list as the starting point for each of the 500 middle loop passes.
6. Not as random, but "random enough" according to the client.

Touch the Big Data Once Only

Do 500 passes over the big data with *one* sequential read by swapping the middle and inner loops. (Actually 500 / number of cores)

```
def compute_kernel(data, sums, bincounts, starters):
  for i in range(len(data)): # Middle loop
    row = data[i]
    t_or_f_sums = sums[0 if some_test(row) else 1]
    for j in range(500): # Inner loop
      bincount = bincounts[starters[j] + i]
      t_or_f_sums[j] += row * bincount

def compute_wrapper(data):
  sums = zip(*[ [[], []] for i in range(500)])
  bincounts = radix_sort(random_array(len(data)))
  bincounts.extend(bincounts)
  starters = [random.randint(0, len(data)) for i in range(500)]

  compute_kernel(data, sums, bincounts, starters)

  cursor.insert(generate_sql(sums[0]))
  cursor.insert(generate_sql(sums[1]))
```

vector * *scalar* is really an “inner, inner loop”

Hand-optimize the C code from Cython

1. Cythonize the compute kernel
2. Hand-optimize 62 lines of C code
3. Permute column summing order to help the L1D cache

Consulting help: Continuum Analytics, Austin

```
if (v_bincount == 0) continue;
double * sumsP = NULL;

// Choose the _T or _F output array.
sumsP = (compute_kernel_DTYPE_t *)(char *)(
        (v_fields->t123 == (*BufPtrStrided1d(
                           compute_kernel_VARIANT_DTYPE_t *,
                            pybuffernd_t123s.rcbuffer->pybuffer.buf,
                            v_t123_index,
                            pybuffernd_t123s.diminfo[0].strides))
         ) ?  pybuffernd_sums_T.rcbuffer->pybuffer.buf :
              pybuffernd_sums_F.rcbuffer->pybuffer.buf);

// Choose the output array row.
sumsP += v_t123_index * (compute_kernel_uniques + 1);

sumsP += (v_bincount == 1) ? v_fields : v_fields * v_bincount
```

vector += vector * scalar is not C! See next slide…

The Permuted Vector Code

```
    /* FIXME: [Performance] Use vector registers for bulk load, multiply/accumulate, and store
                            in full cache lines. --D.S., 2-May-2012 */
   /* Permuted column order to stop hitting the same cache line in rapid succession. */
   if (v_bincount == 1) {
     sumsP[ 0] += v_fields->a1;
     sumsP[ 4] += v_fields->a5;
     sumsP[ 8] += v_fields->a9;
     sumsP[12] += v_fields->a13;
     sumsP[16] += 1.0;
     sumsP[ 1] += v_fields->a2;
     sumsP[ 5] += v_fields->a6;
     sumsP[ 9] += v_fields->a10;
     sumsP[13] += v_fields->a14;
     …

   } else {
     sumsP[ 0] += v_bincount * v_fields->a1;
     sumsP[ 4] += v_bincount * v_fields->a5;
     sumsP[ 8] += v_bincount * v_fields->a9;
     sumsP[12] += v_bincount * v_fields->a13;
     sumsP[16] += v_bincount;
     sumsP[ 1] += v_bincount * v_fields->a2;
     sumsP[ 5] += v_bincount * v_fields->a6;
     sumsP[ 9] += v_bincount * v_fields->a10;
     sumsP[13] += v_bincount * v_fields->a14;
     …
   }
 }
```

The Permuted Vector Code

Twitter engineer: “You can’t write reliable code in C.”

```
/* FIXME: [Performance] Use vector registers for bulk load, multiply/accumulate, and store
...
if (v_bincount == 1) {
  sumsP[ 0] += v_fields->a1;
  ...
  sumsP[12] += v_fields->a13;
  sumsP[16] += 1.0;
  sumsP[ 1] += v_fields->a2;
  sumsP[ 5] += v_fields->a6;
  sumsP[ 9] += v_fields->a10;
  sumsP[13] += v_fields->a14;
  …

} else {
  sumsP[ 0] += v_bincount * v_fields->a1;
  sumsP[ 4] += v_bincount * v_fields->a5;
  sumsP[ 8] += v_bincount * v_fields->a9;
  sumsP[12] += v_bincount * v_fields->a13;
  sumsP[16] += v_bincount;
  sumsP[ 1] += v_bincount * v_fields->a2;
  sumsP[ 5] += v_bincount * v_fields->a6;
  sumsP[ 9] += v_bincount * v_fields->a10;
  sumsP[13] += v_bincount * v_fields->a14;
  …
 }
}
```

The Permuted Vector Code

Twitter engineer: “You can’t write reliable code in C.”

That’s dumb. Use the right tool for the job.

```
    sumsP[12] += v_fields->a13;
    sumsP[16] += 1.0;
    sumsP[ 1] += v_fields->a2;
    sumsP[ 5] += v_fields->a6;
    sumsP[ 9] += v_fields->a10;
    sumsP[13] += v_fields->a14;

    sumsP[ 1] += v_bincount * v_fields->a2;
    sumsP[ 5] += v_bincount * v_fields->a6;
    sumsP[ 9] += v_bincount * v_fields->a10;
    sumsP[13] += v_bincount * v_fields->a14;
    …
  }
}
```

Future (Potential) Speedups

1. Use faster hardware: more cores, more cache, more GHz
2. Replace bit-valued byte columns with one bit-masked column to cut row width from 36 to 30 bytes
3. Use CPU vector instructions (AVX2, FMA, etc.)

   Example: Load/add/store the four floats with 3 instructions, not 12

   e.g. `VEC_LOAD_FLOAT4; VEC_ADD_FLOAT4; VEC_STORE_FLOAT4`

   → Parallelizes computation within the core

   → Relieves L1D cache pressure
4. Rewrite the compute kernel in assembler
5. Use Linux API calls to bind RAM allocation by socket
6. Port to GPU/LRB using the GPU library, then primitives
7. Clusterize

However…

“Good enough is perfect.”

Analyzer runs in 5-10 minutes.
Fetcher takes 15-20 hours.
Pretty-format report refreshes at 9 am.

Fast Fetcher

- Parallelized using a foreman/worker model
- 6 process slots cut runtime to 2 hours
- Fully parallel crashes the database
- Wide variation in run times, so static parallelism wouldn't be appropriate

Future Fast Fetcher

1. Load directly to numpy array
2. Improve RDBMS query speed (2nd attempt)
3. Overlap query (IO), data massage (CPU)
4. Speed up data massage
5. Cache previous day's data
6. Switch from batch to on-line architecture

Pretty Printer

- Batch job was finishing after the 9 am Pretty Print

Improvements:

- Speed up Fetcher
- Re-run several times each day (HACK!)
- Analyzer or Writer triggers Pretty Printer

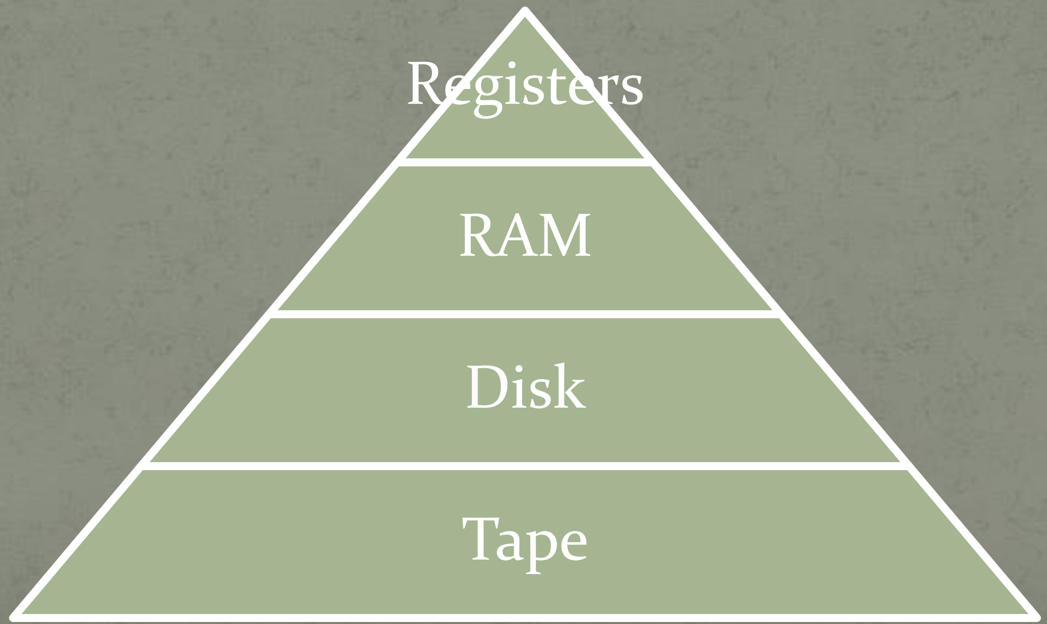
Summary

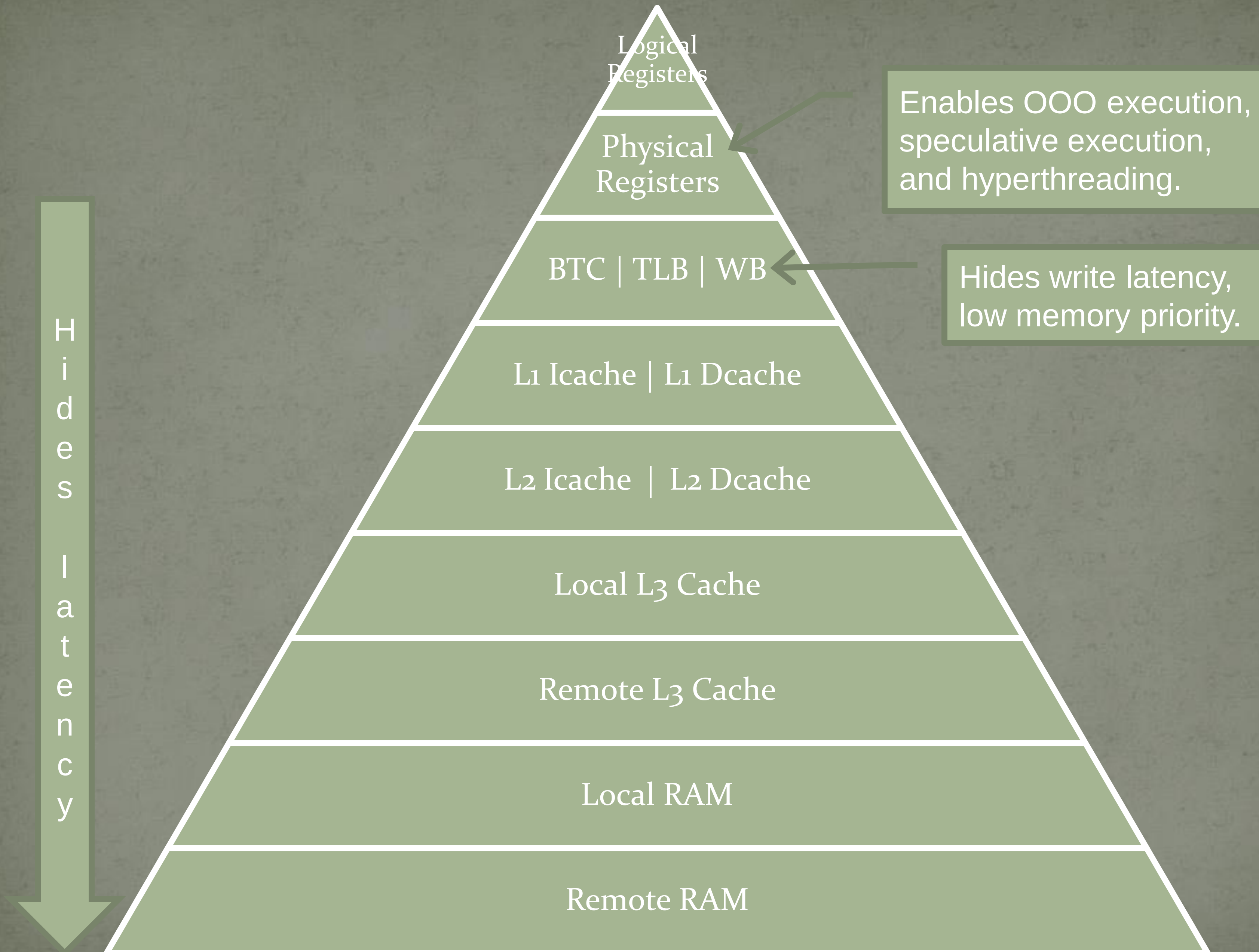
1. Know the goal
2. Identify the bottleneck
3. Build and use a quick test environment
4. Use appropriate model(s) of parallelism
   In the present case, pipelining and shared memory
5. Verify proper use of language extensions
6. Your intuition is wrong; turn it off

Part II: Lessons Learned

- Understanding RAM: How is RAM like a disk?
- Architecture: Von Neumann doesn't live here anymore.
- Scalability is for Suckers: You're not Google.
- Batch jobs aren't "ironic." They're just lame.
- Use sharp tools for cutting, not pounding dirt.

Memory Hierarchy



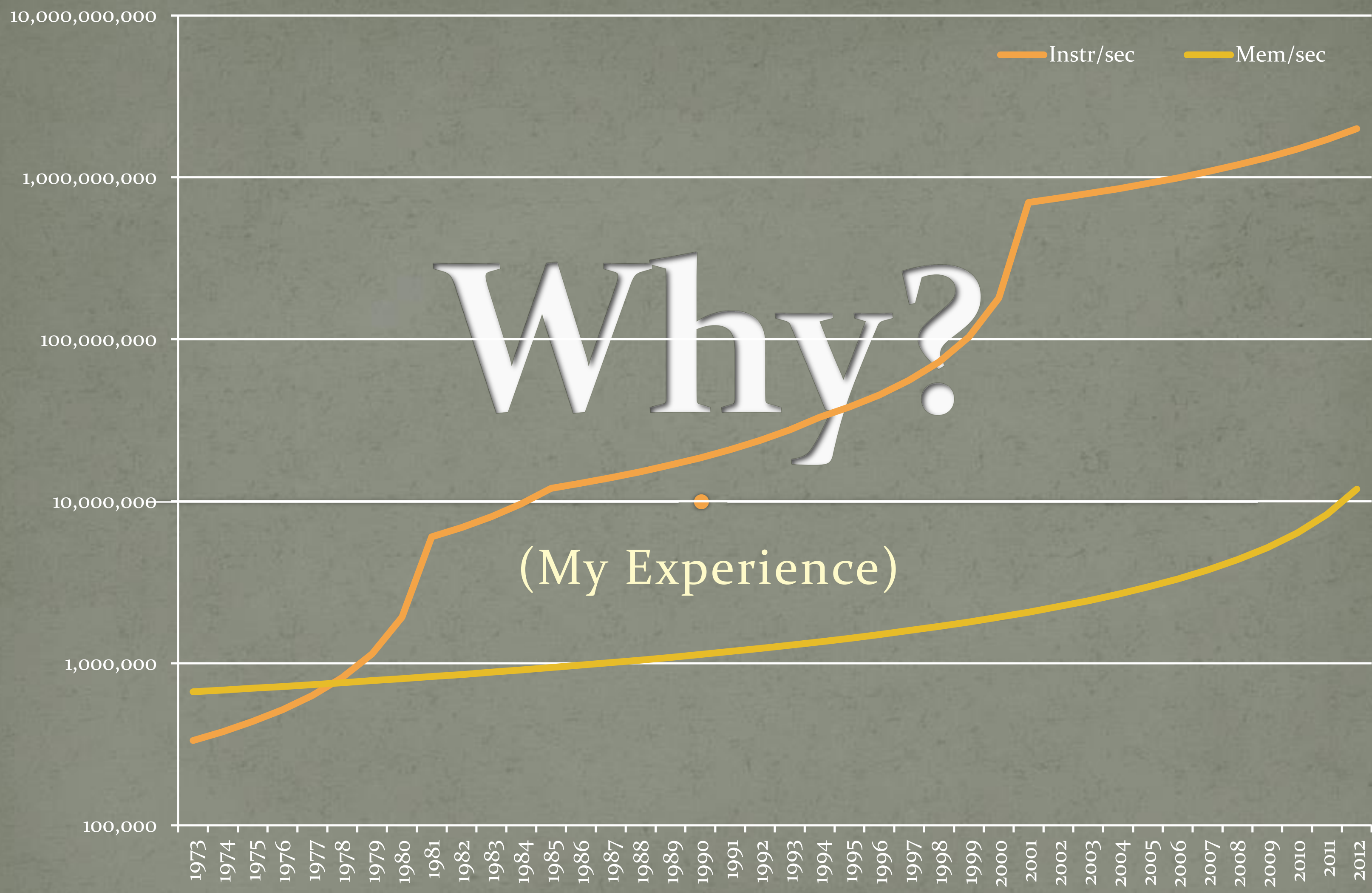


davidschachter@gmail.com

Why is Latency an Issue?

Why?

(My Experience)



RAM/OOO Summary

1. RAM is not "Random Access Memory"
   Avoid reads, esp. random reads, and all writes
   Access vectors in L1D cache-friendly blocks
2. Use HW primitives (int, float), not VM *fuzzballs*
3. Write OOO/cache-friendly code, typically in C
4. Touch the big data exactly once

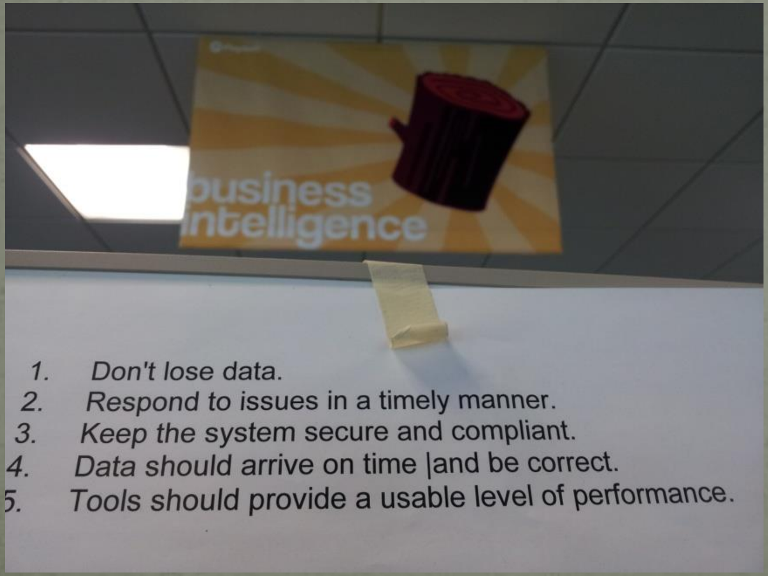
Speed has Value

100X Faster = 100X Fewer Servers

\$300k/mo → \$3k/mo?

Lower CapEx, OpEx, and CO^2

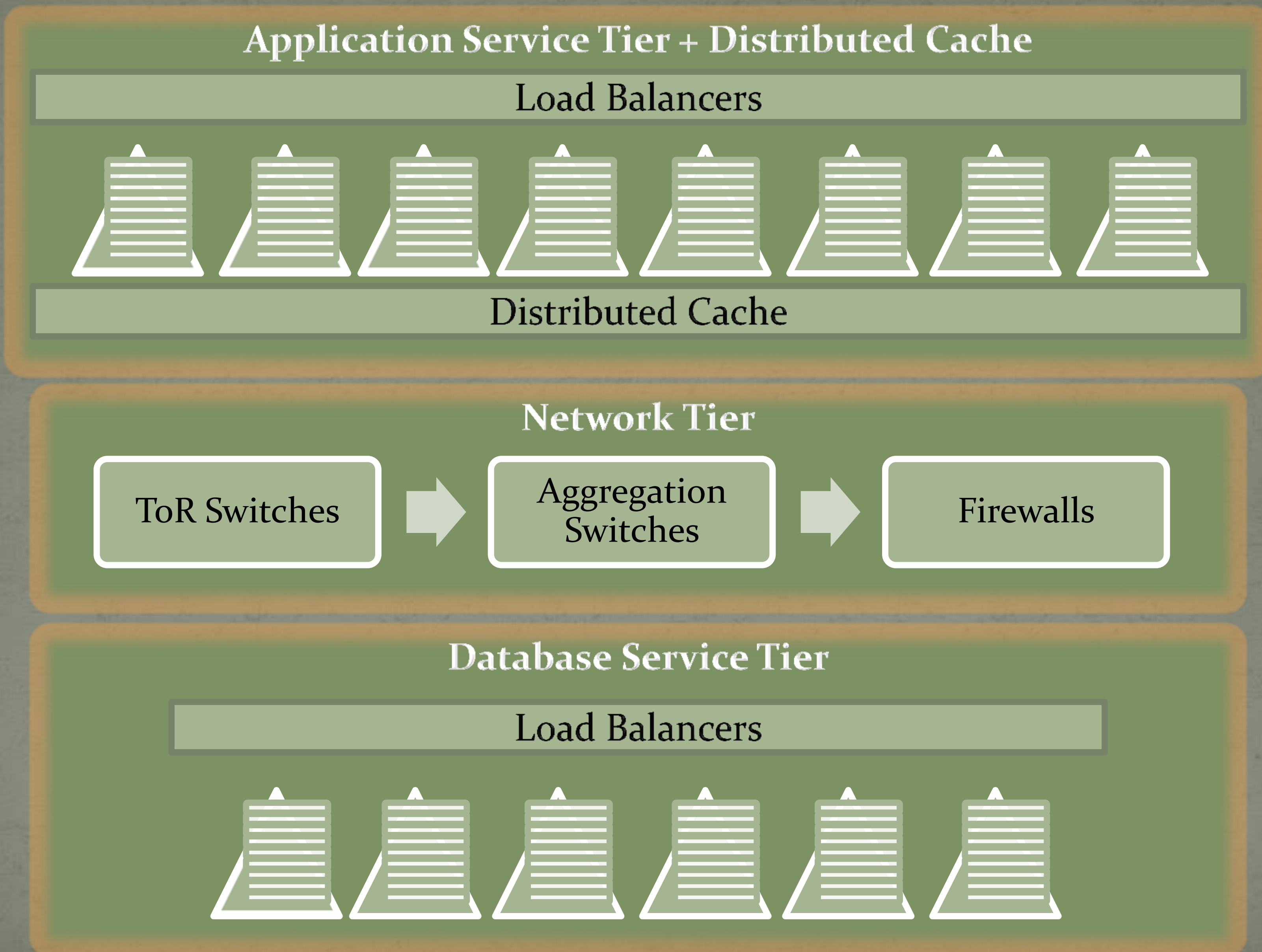
Clusters are Hard



Cluster management is hard.

We have proven this.

1. OOD config files → Data loss
2. OOD SW → Slow crash recovery
3. Sprawl → Bad security
4. Complexity → Late data, reruns
5. Cluster DBMS → Low speed


Application Service Tier + Distributed Cache
Load Balancers
Distributed Cache
Network Tier
ToR Switches
Aggregation Switches
Firewalls
Database Service Tier
Load Balancers

High Availability Server Configuration

Controlled Power or Network Switch

Network Tier

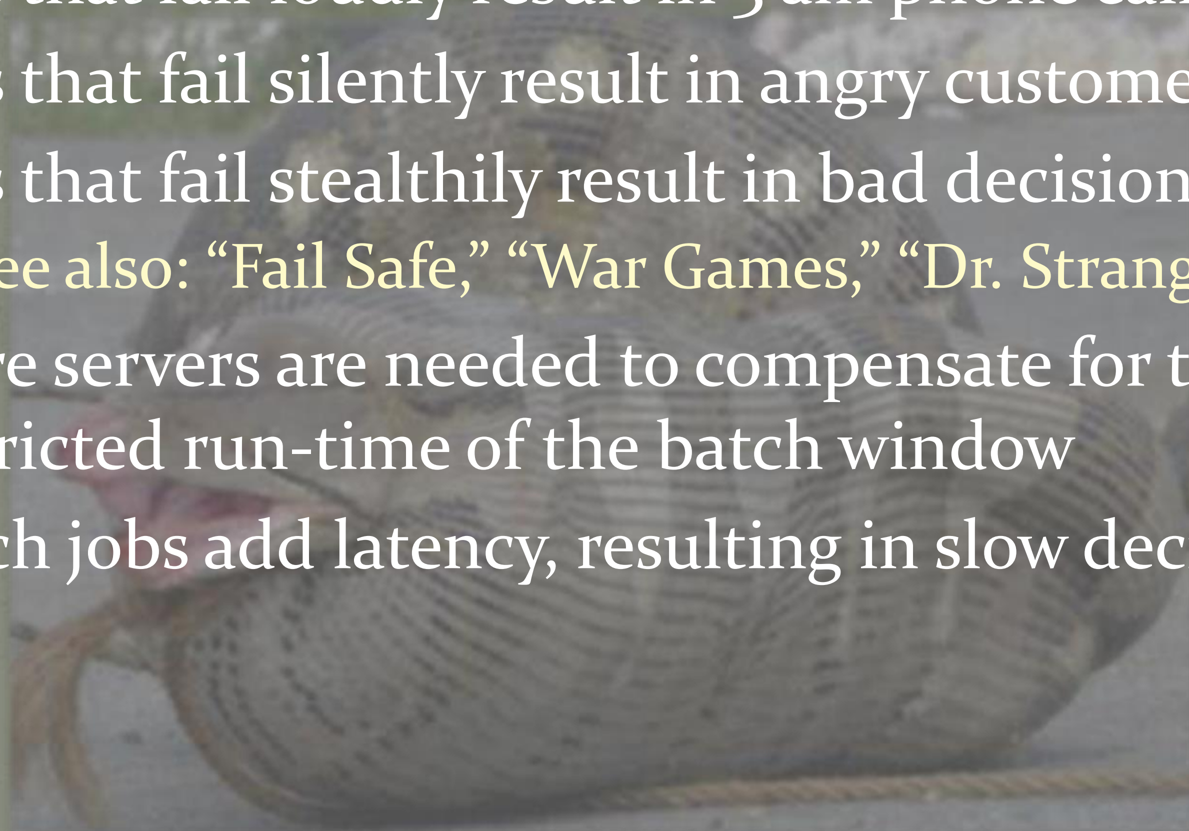
Firewall

Batching: A Pig in a Python

Source: http://fractalbox.files.wordpress.com/2009/06/python_1.jpg

Batching: A Pig in a Python

- Jobs that fail loudly result in 3 am phone calls
- Jobs that fail silently result in angry customers
- Jobs that fail stealthily result in bad decisions
  - See also: “Fail Safe,” “War Games,” “Dr. Strangelove”
- More servers are needed to compensate for the restricted run-time of the batch window
- Batch jobs add latency, resulting in slow decisions



Source: http://fractalbox.files.wordpress.com/2009/06/python_1.jpg

Summary

Fast code is cheap code.
Understand and unleash your machine. It is incredibly fast.
Speed up your code first. Clusterize as a last resort.
If your business is real-time, your software should be, too.
Make sure your tools are working correctly.

One More Thing…

40 year emphasis on productivity at what cost?

One More Thing…

40 year emphasis on productivity at what cost?

- "Programmer productivity is 7-12 lines per day."
- Assembler → Compiler → OOP → VM (Dynamic)
- Bare metal → Batch OS → Time sharing → Hypervisor
- No tools → `ed`, `lint`, `db`, `sccs` → CASE tools

Efficient code is a competitive advantage

Scalability is For Suckers. Performance is King.

Thank you

How?

How to address performance problems?

“Hi, Mr. Performance Problem?”

NO.

Know the Goal

How do you know when you're done?

Find the Bottleneck

1. External resources (SAN, Web, …)
2. Local I/O (disk, tape, …)
3. CPU
4. RAM (paging, latency, bandwidth)

Consider Solutions

1. Faster hardware (easy and quick)
2. Smarter software (takes time and testing)
3. What, me worry? (How many Microsoft CSEs does it take to change a light bulb?)
4. Clusterize (hard to design, build, operate)

RAM Lessons

1. Access sequentially, not randomly
   → Uses optimized access modes
2. Bulk writes thrash the WB; avoid them.
   → Break vector ops into L1D cache blocks