DataFlow

Windowing: where in event time data are grouped

Triggering: when in processing time groups are emitted

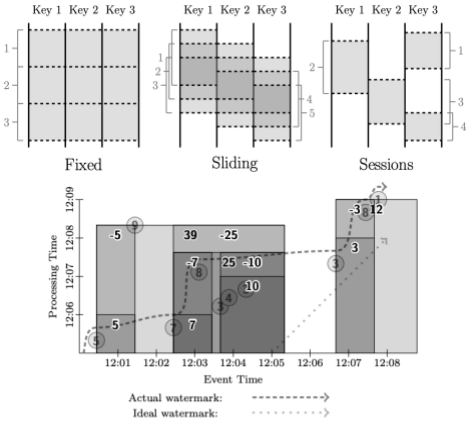


Figure 14: Sessions, Retracting

In this example, we output initial singleton sessions for values 5 and 7 at the first one-minute processing-time boundary. At the second minute boundary, we output a third session with value 10, built up from the values 3, 4, and 3. When the value of 8 is finally observed, it joins the two sessions with values 7 and 10. As the watermark passes the end of this new combined session, retractions for the 7 and 10 sessions are emitted, as well as a normal datum for the new session with value 25. Similarly, when the 9 arrives (late), it joins the session with value 5 to the session with value 25. The repeated watermark trigger then immediately emits retractions for the 5 and the 25, followed by a combined session of value 39. A similar dance occurs for the values 3, 8, and 1, ultimately ending with a retraction for an initial 3 session, followed by a combined session of 12.

Naiad

No support for stale update

Checkpointing needs pause all workers and flush message queues

This restricted looping structure allows us to design logical timestamps based on the dataflow graph structure. Every message bears a logical timestamp of type

$$\text{Timestamp} : \overbrace{(e \in \mathbb{N},}^{\text{epoch}} \overbrace{\langle c_1, \ldots, c_k \rangle \in \mathbb{N}^k)}^{\text{loop counters}}$$

where there is one loop counter for each of the k loop contexts that contain the associated edge. These loop counters explicitly distinguish different iterations, and allow a system to track forward progress as messages circulate around the dataflow graph.

The ingress, egress, and feedback vertices act only on the timestamps of messages passing through them. The vertices adjust incoming timestamps as follows:

Vertex	Input timestamp	Output timestamp
Ingress	$(e, \langle c_1, \ldots, c_k \rangle)$	$(e, \langle c_1, \ldots, c_k, 0 \rangle)$
Egress	$(e, \langle c_1, \ldots, c_k, c_{k+1} \rangle)$	$(e, \langle c_1, \ldots, c_k \rangle)$
Feedback	$(e, \langle c_1, \ldots, c_k \rangle)$	$(e, \langle c_1, \ldots, c_k + 1 \rangle)$

For two timestamps $t_1 = (x_1, \vec{c}_1)$ and $t_2 = (x_2, \vec{c}_2)$ within the same loop context, we order $t_1 \leq t_2$ if and only if both $x_1 \leq x_2$ and $\vec{c}_1 \leq \vec{c}_2$, where the latter uses the lexicographic ordering on integer sequences. This order corresponds to the constraint on future times at which one message could result in another, a concept that we formalize in the following subsections.

2.2 Vertex computation

Timely dataflow vertices send and receive timestamped messages, and may request and receive notification that they have received all messages bearing a specific timestamp. Each vertex v implements two callbacks:

v.OnRecv(e : Edge, m : Message, t : Timestamp)
v.OnNotify(t : Timestamp).

A vertex may invoke two system-provided methods in the context of these callbacks:

this.SendBy(e : Edge, m : Message, t : Timestamp)
this.NotifyAt(t : Timestamp).

Each call to u.SendBy(e,m,t) results in a corresponding invocation of v.OnRecv(e,m,t), where e is an edge from u to v, and each call to v.NotifyAt(t) results in a corresponding invocation of v.OnNotify(t). The invocations of OnRecv and OnNotify are queued, and for the most part the model is flexible about the order in which they may be delivered. However, a timely dataflow system must guarantee that v.OnNotify(t) is invoked only after no further invocations of v.OnRecv(e,m,t'), for $t' \leq t$, will occur. v.OnNotify(t) is an indication that all v.OnRecv(e,m,t) invocations have been delivered to

Spark Streaming

Unify Batch & Interactive Processing

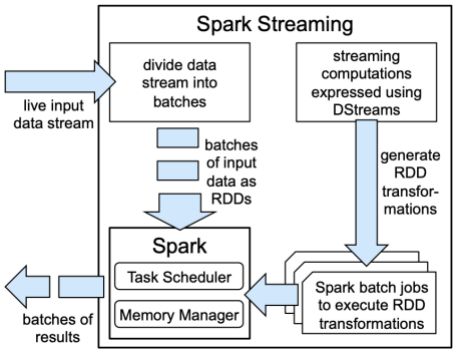
Use speculative execution for straggler

Lineage for parallel workers recovery

Connect to new master when old fails

Fine to lose some running tasks on reconnect

Based on immutable RDD



D-Streams provide a *track* operation that transforms streams of (Key, Event) records into streams of (Key, State) records based on three arguments:

- An *initialize* function for creating a State from the first Event for a new key.
- An *update* function for returning a new State given an old State and an Event for its key.
- A *timeout* for dropping old states.

Windowing: The *window* operation groups all the records from a sliding window of past time intervals into one RDD. For example, calling `words.window("5s")` in the code above yields a D-Stream of RDDs containing the words in intervals $[0,5)$, $[1,6)$, $[2,7)$, etc.

Incremental aggregation: For the common use case of computing an aggregate, like a count or max, over a sliding window, D-Streams have several variants of an incremental *reduceByWindow* operation. The simplest one only takes an associative merge function for combining values. For instance, in the code above, one can write:

```
pairs.reduceByWindow("5s", (a, b) => a + b)
```

This computes a per-interval count for each time interval only once, but has to add the counts for the past five seconds repeatedly, as shown in Figure 4(a). If the aggregation function is also *invertible*, a more efficient version also takes a function for "subtracting" values and maintains the state incrementally (Figure 4(b)):

```
pairs.reduceByWindow("5s", (a,b) => a+b, (a,b) => a-b)
```

PowerGraph

Fine-grained parallelism

Not all vertices are activated

Vertex-cuts more suitable with power rule

Quickly shatter a graph by cutting a small fraction of the very high degree vertices

Checkpoint at the end of super-step for sync

No straggler mitigation

(a) Edge-Cut (b) Vertex-Cut

Because the greedy-heuristic is a de-randomization it is guaranteed to obtain an expected replication factor that is no worse than random placement and in practice can be much better. Unlike the randomized algorithm, which is embarrassingly parallel and easily distributed, the greedy algorithm requires coordination between machines. We consider two distributed implementations:

Coordinated: maintains the values of $A_i(v)$ in a distributed table. Then each machine runs the greedy heuristic and periodically updates the distributed table. Local caching is used to reduce communication at the expense of accuracy in the estimate of $A_i(v)$.

Oblivious: runs the greedy heuristic independently on each machine. Each machine maintains its own estimate of A_i with no additional communication.

```
interface GASVertexProgram(u) {
  // Run on gather_nbrs(u)
  gather(D_u, D_(u,v), D_v) → Accum
  sum(Accum left, Accum right) → Accum
  apply(D_u, Accum) → D_u^new
  // Run on scatter_nbrs(u)
  scatter(D_u^new, D_(u,v), D_v) → (D_(u,v)^new, Accum)
}
```

Algorithm 1: Vertex-Program Execution Semantics

Input: Center vertex u
if *cached accumulator* a_u *is empty* **then**
  foreach *neighbor* v *in gather_nbrs(u)* **do**
    $a_u \leftarrow \text{sum}(a_u, \text{gather}(D_u, D_{(u,v)}, D_v))$
  end
end
$D_u \leftarrow \text{apply}(D_u, a_u)$
foreach *neighbor* v *scatter_nbrs(u)* **do**
  $(D_{(u,v)}, \Delta a) \leftarrow \text{scatter}(D_u, D_{(u,v)}, D_v)$
  if a_v *and* Δa *are not Empty* **then** $a_v \leftarrow \text{sum}(a_v, \Delta a)$
  else $a_v \leftarrow$ Empty
end

PageRank

```
// gather_nbrs: IN_NBRS
gather(D_u, D_(u,v), D_v):
  return D_v.rank / #outNbrs(v)
sum(a, b): return a + b
apply(D_u, acc):
  rnew = 0.15 + 0.85 * acc
  D_u.delta = (rnew - D_u.rank)/
              #outNbrs(u)
  D_u.rank = rnew
// scatter_nbrs: OUT_NBRS
scatter(D_u, D_(u,v), D_v):
  if(|D_u.delta|>ε) Activate(v)
  return delta
```

Greedy Graph Coloring

```
// gather_nbrs: ALL_NBRS
gather(D_u, D_(u,v), D_v):
  return set(D_v)
sum(a, b): return union(a, b)
apply(D_u, S):
  D_u = min c where c ∉ S
// scatter_nbrs: ALL_NBRS
scatter(D_u, D_(u,v), D_v):
  // Nbr changed since gather
  if(D_u == D_v)
    Activate(v)
  // Invalidate cached accum
  return NULL
```

GraphX

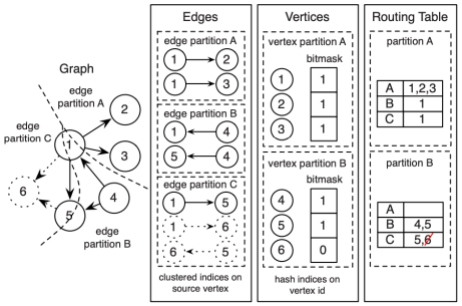
Immutable data
Overhead of distribution is high
Join strategy: send vertices to the edge site
Automatic Join Elimination

```
def Gather(a: Double, b: Double) = a + b
def Apply(v, msgSum) {
  PR(v) = 0.15 + 0.85 * msgSum
  if (converged(PR(v))) voteToHalt(v)
}
def Scatter(v, j) = PR(v) / NumLinks(v)
```

```
CREATE VIEW triplets AS
SELECT s.Id, d.Id, s.P, e.P, d.P
FROM edges AS e
JOIN vertices AS s JOIN vertices AS d
ON e.srcId = s.Id AND e.dstId = d.Id
```

The `mrTriplets` (Map Reduce Triplets) operator encodes the essential two-stage process of graph-parallel computation defined in Section 3.2. Logically, the `mrTriplets` operator is the composition of the *map* and *group-by* dataflow operators on the triplets view. The user-defined map function is applied to each triplet, yielding a value (i.e., a message of type `M`) which is then aggregated at the destination vertex using the user-defined binary aggregation function as illustrated in the following:

```
SELECT t.dstId, reduceF(mapF(t)) AS msgSum
FROM triplets AS t GROUP BY t.dstId
```



Multicast Join: While *broadcast join* in which all vertices are sent to each edge partition would ensure joins occur on edge partitions, it could still be inefficient since most partitions require only a small subset of the vertices to complete the join. Therefore, GraphX introduces a *multicast join* in which each vertex property is sent only to the edge partitions that contain adjacent edges. For each vertex GraphX maintains the set of edge partitions with adjacent edges. This join site information is stored in a *routing table* which is co-partitioned with the vertex collection (Figure 3). The routing table is associated with the edge collection and constructed lazily upon first instantiation of the triplets view.

Weld

Hard to debug and scale to more libraries
Fully deterministic, cannot express asnyc algo
Falut tolerance restricted to a single machine
Primitives: scalar, structure, vector, dict
Builder: vecbuilder, merger, etc.

Builders support three basic operations. `merge(b, v)` adds a new value `v` into the builder `b` and returns a new builder[2] to represent the result. Merges into builders are associative, allowing them to be reordered. `result(builder)` destroys the builder and returns its final result: no further operations are allowed on it after this call. Finally, the `for(vector, builders, func)` operator applies a function of type `(builders, T) => builders` to each element of a vector in parallel, updating one more builders for each one, and returns the final set of builders. The `for` operator is the only way to launch parallel work in Weld: the iterations of the loop will run in parallel and merge their results into the provided builders.

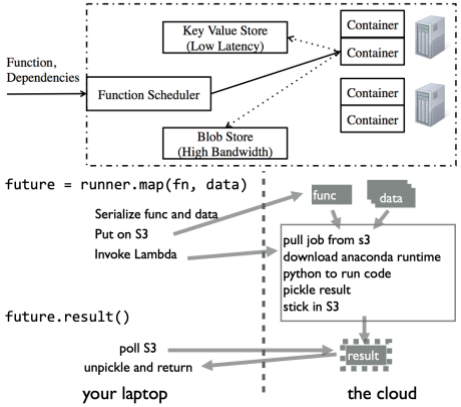
Optimizations Passes	
Loop Fusion	Fuses adjacent loops to avoid materializing intermediate results when the output of one loop is used as the input of another. Also fuses multiple passes over the same vector.
Size Analysis	Infers the size of output vectors statically.
Loop Tiling	Breaks nested loops into blocks to exploit caches by reusing values faster [8].
Vectorization & Predication	Transforms loops with simple inner bodies to use vector instructions. Branches inside the loop body are transformed into unconditional select instructions (predication).
Common Subexpression Elimination	Transforms the program to not run the same computation multiple times.

```
// Before horizontal fusion
v1 := result(for(
    v0, vecbuilder[int], (b,i,x) => merge(b,x+1)))
v2 := result(for(
    v0, merger[int,+], (b,i,x) => merge(b,x)))
{v1, v2}

// After horizontal fusion
tmp := for(v0, {vecbuilder[int], merger[int,+]},
    (bs,i,x) => {merge(bs.0, x+1), merge(bs.1, x)}
)
{result(tmp.0), result(tmp.1)}
```

PyWren

Breakdown computation into stateless funcs
Schedule on serverless containers
Use external storage for state management
No need for explicit fault tolerance
More elastic. Good for streaming workload
Overhead from fetching state. Cold start
Network bandwidth is pretty good
Blob storages more suitable for large file
Scheduler might be the bottleneck
Information gap between user and server
No sharing inventive



PyWren serializes a Python function using `cloudpickle` [7], capturing all relevant information as well as most modules that are not present in the server runtime[3]. This eliminates the majority of user overhead about deployment, packaging, and code versioning. We submit the serialized function along with each serialized datum by placing them into globally unique keys in S3, and then invoke a common Lambda function. On the server side, we invoke the relevant function on the relevant datum, both extracted from S3. The result of the function invocation is serialized and placed back into S3 at a pre-specified key, and job completion is signaled by the existence of this key. In this way, we are able to reuse one registered Lambda function to execute different user Python functions and mitigate the high latency for function registration, while executing functions that exceed Lambda's code size limit.

TPU

No features to improve the average case
No caches, branch pred, out-of-order execution
Simple design with MACs, Unified Buffer
TPU is not energy proportional
systolic execution to save energy by reducing reads and writes of the Unified Buffer

To illustrate the performance of the six apps on the three processors, we adapt the Roofline Performance model from high-performance computing (HPC) [58]. This simple visual model is not perfect, yet it offers insights into the causes of performance bottlenecks. The assumption behind the model is that applications don't fit in on-chip caches, so they are either computation-limited or memory bandwidth-limited. For HPC, the Y-axis is performance in floating-point operations per second, thus the peak computation rate forms the "flat" part of the roofline. The X-axis is operational intensity, measured as floating-point operations per DRAM byte accessed. Memory bandwidth is bytes per second, which turns into the "slanted" part of the roofline since (FLOPS/sec)/ (FLOPS/Byte) = Bytes/sec. Without sufficient operational intensity, a program is memory bandwidth-bound and lives under the slanted part of the roofline.

highlights:

- Inference apps usually emphasize response-time over throughput since they are often user facing.
- As a result of latency limits, the K80 GPU is just a little faster for inference than the Haswell CPU, despite it having much higher peak performance and memory bandwidth.
- While most architects are accelerating CNNs, they are just 5% of our datacenter workload.
- The TPU is about 15X – 30X faster at inference than the K80 GPU and the Haswell CPU.
- Four of the six NN apps are memory bound; if the TPU were revised to have the same memory as the K80 GPU, it would be about 30X – 50X faster than the GPU and CPU.
- Despite having a much smaller and lower power chip, the TPU has 25 times as many MACs and 3.5 times as much on-chip memory as the K80 GPU.
- The performance/Watt of the TPU is 30X – 80X that of its contemporary CPUs and GPUs; a revised TPU with K80 memory would be 70X – 200X better.