F^2: Reenvisioning Data Analytics Systems

Robert Grandl†, Arjun Singhvi, Raajay Viswanathan°, Aditya Akella
University of Wisconsin - Madison, †Google, °Uber

under review

***Abstract*—** Today's data analytics frameworks are intrinsically compute-centric. Key details of analytics execution – work allocation to distributed compute tasks, intermediate data storage, task scheduling, etc. – depend on the pre-determined physical structure of the high-level computation. Unfortunately, this hurts flexibility, performance, and efficiency. We present F^2, a new analytics framework that cleanly separates computation from intermediate data. It enables runtime visibility into data via programmable monitoring, and data-driven computation (where intermediate data values drive when and what computation runs) via an event abstraction. Experiments with an F^2 prototype on a large cluster using batch, streaming, and graph analytics workloads show that it significantly outperforms state-of-the-art compute-centric engines.

1 Introduction

Many important applications in diverse settings (e.g., health care, cybersecurity, education, and IoT) rely on analyzing large volumes of data which can include archived relational tables, event streams, graph data, etc. To ease analysis of such diverse large data sets, several analytics frameworks have emerged [43, 18, 44, 35, 26, 36, 10, 32, 4, 38, 39]. These enable data parallel computation, where a job's analysis logic is run in parallel on data shards spread across multiple machines in large clusters.

Almost all these frameworks, be they for batch [23, 43, 18], stream [44, 4] or graph processing [35, 26, 36], have their intellectual roots in MapReduce [23], a time-tested execution framework for data parallel workloads. While they have many differences, existing frameworks share a key attribute with MapReduce, in that they are *compute-centric*. Their focus, like MapReduce, is on splitting a given job's computational logic and distributing it across compute units, or *tasks*, to be run in parallel. Like MapReduce, all aspects of the subsequent execution of the job are rooted in the job's computational logic and its task-level distribution, i.e., job structure (§2). These include the fact that the logic running inside tasks, and task parallelism, are static and/or predetermined; intermediate data is partitioned as it is generated, and routed to where it is consumed, as a function of the original compute structure; dependent tasks are launched when a fraction of tasks they depend on finish, etc.

Compute-centricity was a natural choice for MapReduce. Knowing job structure beforehand made it easy to understand how to carve computational units to execute tasks. Also, because failures are a common case in large clusters, fault tolerance is important, and compute centricity provided clean mechanisms to recover from failures[1]. Schedulers became simple because of having to deal with static inputs (fixed tasks/dependency structures). Support for fault tolerance and scheduling allowed developers to focus on improving the programming model to support broad applications [5, 3], which ultimately spurred wide adoption of the MapReduce paradigm.

But, is compute-centricity the right choice, especially for the diverse frameworks that emerged from MapReduce? Is it the right choice *today* given that compute infrastructure is vastly different, with advances in computing such as containerization and serverless platforms [6], and in distributed key-value (KV) storage services [9, 41]?

We answer the first question by arguing that compute centricity gets in the way by imposing at least four fundamental limitations (§2): (1) Intermediate data is opaque; there is no way to adapt job execution based on runtime data properties. (2) Static parallelism and intermediate data partitioning constrain adaptation to data skew and resource flux. (3) Tying execution schedules to compute structure can lead to resource waste while tasks wait for input to become available. (4) Compute-based organization of intermediate data can result in storage hotspots and poor cross-job I/O isolation, and it curtails data locality.[2] Thus, while compute-centricity has served us well, it begets inflexibility, poor performance, and inefficiency, hurting current and future key applications.

The above limitations arise from (1) *tight coupling between data and compute*, and (2) *intermediate data being relegated to a second class citizen*. Motivated by these observations, and inspired by modern serverless platforms, we propose a new framework, F^2. It cleanly separates computation from all intermediate data. Intermediate data is written to/read from a separate key-value datastore. The store is programmable – applications can provide custom logic for monitoring runtime properties of data. We introduce an event abstraction that allows the store to signal to an execution layer when an application's intermediate data values satisfy certain properties. Events thus enable

[1] Only tasks on a failed machine needed to be re-executed.

[2] E.g., it is impossible to achieve data-local reduce task placement.

complete data-driven computation in F^2, where, based on runtime data properties, the execution layer decides what logic to launch in order to further process data generated, how many parallel tasks to launch and when/where to launch them, and which resources to allocate to tasks to process specific data. Data-driven computation improves performance, efficiency, isolation, and flexibility relative to all compute-centric frameworks.

In designing F^2, we make the following contributions:

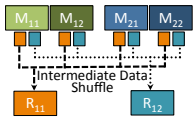
- We present novel and scalable APIs for programmable intermediate data monitoring, and for leveraging events for rich data-driven actions. Our APIs balance expressiveness against overhead and complexity.
- We show how to organize intermediate data from multiple jobs in the datastore so as to achieve per-job data locality and fault tolerance, and cross-job isolation. Since obtaining an optimal data organization is NP-Hard, we develop novel heuristics that carefully trade-off among these storage objectives.
- We develop novel altruism-based heuristics for the job execution layer for data-driven task parallelism and placement. This minimizes both skew in data processed and data shuffle cost, while operating under the constraints of dynamic resource availability. Each task is launched in a container whose size is late-bound to the actual data allocated to the task.

We have built an F^2 prototype by refactoring and adding to Tez [40] and YARN [45] (15000 lines).We currently support batch, graph, and stream processing. We deploy and experiment with our prototype on a 50 machine cluster in CloudLab [7].

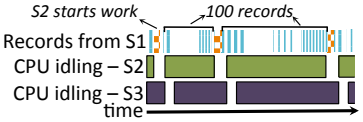
We compare against the state-of-the-art compute centric (CC) approaches. F^2 improves median (95%-ile) job completion time (JCT) by $1.3-1.7\times$ ($1.5-2.2\times$) across batch, streaming, and graph analytics. F^2 reduces idling by launching (the right number of appropriately-sized) tasks only when custom predicates on input data are met, and avoids expensive data shuffles even for consumer tasks. Under high cluster load, F^2 offers $1.8\times$ better JCT than CC due to better cross-job data management and isolation. F^2's data-driven actions enable computation to start much sooner than CC, leading to $1.6\times$ better stream or graph JCTs. Finally, F^2's data-driven logic changes can improve JCT by $1.4\times$.

2 On Eschewing Compute Centricity

Background: Production batch [50, 23, 5], stream [4, 50, 14] or graph [25, 35, 26] analytics frameworks support the execution of multiple inter-dependent *stages* of computation. Each stage is executed simultaneously within different *tasks*, each processing different data shards, or



(a) A batch analytics job. Intermediate data is partitioned into two key ranges, one per reduce task, and stored in local files at map tasks.



(b) Data flow in sample streaming job. Tasks in all stages are always running. Output of a stage is immediately passed to a task in immediate downstream stage. However, CPU is idle until task in Stage 2 receives 100 records after which computation is triggered.

Figure 1: Simplified examples of existing analytics systems. Graph analytics, not shown, is similar: the different stages are iterations in a graph algorithm; thus, all stages execute the same processing logic (the vertex program). Intermediate data generated by one iteration is a set of messages for each graph vertex. This data is moved to the relevant vertex program where it becomes input to the next iteration.

partitions, to generate input for a subsequent stage.

A simple *batch analytics* job is shown in Fig. 1a; here, two tables need to be filtered on provided predicates and joined to produce a new table. The computation has three stages – *two maps* (M_1 and M_2) to filter the tables, and *one reduce* (R_1) to execute the join. Execution of this job proceeds as follows [23, 5]: (1) The map tasks from both the stages execute first, and are run simultaneously (as they have no dependencies) with each task processing a partition of the corresponding input table. (2) Map intermediate results are written to local disk by each task, split into files, one per downstream reduce task. (3) Reduce tasks are launched when the map stages are nearing completion; each reduce task *shuffles* relevant intermediate data from all map tasks' locations and generates output.

A typical *stream analytics* job has a similar computation model [14, 33, 51]. The main difference is that, tasks in all streaming stages are always running. A *graph analytics job*, in a framework that relies on the popular message passing abstraction [35], has a similar, even more simplified model (see Fig. 1 caption).

Compute-centricity: The above frameworks are designed primarily with the goal of splitting up and distributing computation across multiple machines. The composition and structure of this distributed computation is a first class entity. The exact computation in each task is assumed to be known beforehand. The way in which intermediate data is partitioned and routed to consumer tasks, and when and how dependent computation is launched, are tied to compute structure. We use the term *compute-centric* to refer to this design pattern. Here, intermediate data is a strict second class entity.

2.1 Issues with Compute-centricity

The above systems' performance depends on a few factors: (1) the computation logic for each stage which determines task run time, e.g., for joining two large datasets

sort-merge join is more expensive than a hash or broadcast join, (2) stages' parallelism — the higher the #tasks the lesser the amount of work each task has to do, leading to potentially lower stage execution times, (3) the resource allocation and task scheduling algorithms, and (4) the volume of data shuffled between stages, and skew in cross-task data distribution [17]. Improving overall performance requires careful optimization of all these factors. However, existing systems' compute-centricity imposes key constraints on potential optimizations:

(1) Data opacity, and compute rigidity: Being a second class entity, data transferred between stages of computation is treated as an opaque blob; there's no visibility into it. A job cannot programmatically monitor all data produced by any stage (e.g., determining if a certain value v was seen in intermediate data). This prohibits triggering or adapting computation based on run time intermediate data properties (e.g., launch a particular program T in downstream tasks if upstream stage's output satisfies a certain predicate p; else launch T').[3]

Programmable monitoring, and monitoring-driven computation, can help by significantly speeds up job execution. Consider the batch job in Fig. 1a. Existing frameworks determine the type of join in the reduce stage based on coarse statistics [3]; unless one of the tables is small, a sort-merge join is employed to avoid out-of-memory (OOM) errors. However, if per-key histograms of intermediate data are available, we can dynamically determine the type of join to use for *different* reduce tasks. A task can use hash join if the total size of *its* key range is less than the available memory, and merge join otherwise.

(2) Static data plane, and inadaptability to resources/skew: We use the term *data plane* to refer to a job's tasks, their interconnecting dependency edges, and the data partitioning strategy that controls how intermediate data flows on those edges. Current frameworks determine the data plane *statically, prior to job execution*. E.g., in Spark [50] the number of tasks in a stage is determined by the user application or the SparkSQL [13] framework. A hash partitioner is used to place an intermediate (k, v) pair into one of the $|tasks|$ buckets. Pregel [35] vertex-partitions the input graph; partitions do not change during the execution of the graph algorithm.

Static data planes impact performance. First, work allocation to tasks cannot adapt to run time changes. A currently running stage cannot utilize newly available compute resources[4] and dynamically increase its parallelism. Second, work allocation cannot adapt to runtime data skew which can be hard to predict. If some key (or some vertex program) in a partition has an abnormally large number of records (or messages) to process then the corresponding task is significantly slowed down [17].

If parallelism and partitioning strategies can be altered based on intermediate data, we can optimally adapt to resource changes and data skew. Realizing such *flexible data planes*, however, is difficult today due to data opacity. It is also complicated by the coupling of intermediate data to producer tasks: repartitioning data across all producers to align with changes to downstream parallelism is onerous.

(3) Idling due to compute-driven scheduling: Modern cluster schedulers [45, 24] decide when to launch tasks for a stage based on the static task-level structure of a job. When a stage's computation is commutative+associative, schedulers launch its tasks once 90% of all tasks in upstream stages complete [5]. However, the remaining 10% producers can take long to complete (e.g., due to data skew) resulting in tasks idling.

Idling is more common in the streaming setting where downstream tasks are continuously waiting for data from upstream tasks. Consider the example in Fig. 1b: the task in Stage 2 computes and outputs the median for every 100 records received. Thus, it idles, as it can't emit any output until it has received 100 records from Stage 1; as a result the task in Stage 3 *also* idles for a long time. A similar situation arises in graph processing where a task waits for all messages from the previous iteration before proceeding with computation for the current iteration.

Ideally, tasks should be scheduled *only when relevant input is available*. This depends both on the computation logic and the data it consumes. In the above example, computation should be launched only after ≥ 100 records have been generated by an upstream task. As another example, if computation is commutative+associate, it is reasonable to "eagerly" launch tasks to process intermediate data as long as enough data has been generated to process in one batch; also, such a task should be able to exit after it is done processing the current batch. Today, tasks linger to process all data preassigned to them.

Scheduling in the above fashion is difficult today due to the lack of suitable APIs for programmable monitoring (to identify whether relevant data has been generated) and for monitoring-driven computation (to then launch additional computation appropriate for the data).

(4) Second-class data, and storage inefficiencies: Another key issue is where to schedule tasks. Because second-class intermediate data is spread across producer tasks' locations (Fig. 1a), it is *impossible to place con-*

[3] Some frameworks [5] only collect coarse information, such as the final intermediate data file sizes, which they use to avoid launching a task in case relevant upstream files that form task input are empty.

[4] this could happen, e.g., when the job is running on VMs derived from a spot market, or due to large-job departures in a cluster.

sumer asks in a data-local fashion. Such tasks are placed at random [23] and forced to shuffle data for processing. Shuffles substantially inflate job runtimes (∼30% [22]).

Storage inefficiencies also arise from lack of isolation across jobs. Today, tasks are scheduled where appropriate compute units are available [24, 8]. Unfortunately, when tasks from multiple jobs are collocated, it becomes difficult to isolate their hard-to-predict runtime intermediate data I/O. Tasks from jobs generating significant intermediate data may occupy much more local storage and I/O bandwidth than those generating less. Multiple jobs' tasks generating data heavily can create storage hotspots.

3 Data-driven Design in F^2

F^2 overcomes the flaws of compute-centricity by reenvisioning analytics stacks based on three principles:

1. Decoupling compute and data: Many of the above issues are rooted in intermediate data being a second class citizen today. Thus, inspired by serverless architectures [6], F^2 cleanly decouples compute from intermediate data (§4). The latter is written to/read from a separate distributed KV datastore, and managed by a distinct data management layer, called the data service (DS). An execution service (ES) manages compute tasks. Because the store manages data from all stages across all jobs, it can, similar to multi-tenant KV stores [9, 41], enforce universal policies to organize data to meet per-job objectives, e.g., locality for *any* stage's tasks, *and* cluster objectives, such as I/O hotspot avoidance and cross-job isolation.

2. Programmable data visibility: The above separation also enables rich, low-overhead, and scalable approaches to gain visibility into *all* of a stage's runtime data. Inspired by programmable network monitoring [48, 37], F^2 allows a programmer to gather custom runtime properties for all intermediate data of any stage of a job, and of values contained therein, via a narrow, well-defined API (§5.1).

3. Data-driven computation using events: Building on data visibility, F^2 provides an API for subscribing to *events* (§5.2) which form the basis for a rich *intermediate data publish-subscribe substrate*. Programmers can define custom predicates on properties of data values for each stage of their computation. F^2 events notify the application when intermediate data satisfies the predicates. Crucially, events help achieve general *data-driven computation*, where properties of intermediate data drive further computation (§6): Specifically (1) Events ensure that computation is launched *only when* data whose keys/-values satisfy relevant properties is available. (2) Coupled with data properties, events help dynamically determine (a) resource adaptation and optimal parallelism selection based on total data volume and skew across keys, and (b) suitable computation logic to apply to available data.

```
1  JOB job = F2.createJob("WordCount");
2  job.addStage("s1", S1Impl.class,
     DS.DEFAULT_TRIGGER.BATCH).addStage("s2",
     S2Impl.class, DS.DEFAULT_TRIGGER.BATCH);
3  job.addDependency("s1", "s2");
4  // Program DS to raise a DataEvent indicating
5  // #words with more than 100 occurrences.
6  Module monitor = job.createModule("s1",
                DS.MONITOR.NUM_ENTRIES, 100);
18 job.addModule(monitor, action);
19 job.submit();
7  // Create an action to trigger runtime changes.
8  Action action = job.createAction() {
9    @Override
10   void run(DataEvent event) {
11    if (event.type == DS.MONITOR.NUM_ENTRIES
12        && event.value == 0) {
13      job.replaceStage(event.stageID,
14      EmptyImpl.class, event.granuleID);
15    job.refresh();
16   }
17 }
```

Figure 2: F^2 user program for a job where the requirement is to print all the words with more than 100 occurrences.

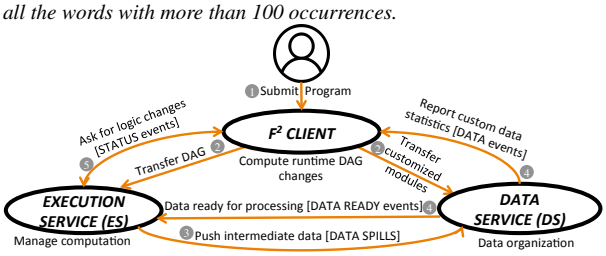


Figure 3: Control flow between the main components of F^2. On receiving data from tasks (of "$s1$"), the DS stores it across granules. DS notifies ES when a granule is ready for further processing (by "$s2$") via an event (step 4). DS sends per-granule collected statistics (number of words with $\geq$ 100 occurrences; line 6 in Fig. 2) to F^2 client via events as well (step 4). Upon receiving this event the ES (step 5) queries the DS for compute logic to be applied to specific granules; here, any granule with $<$ 100 word occurrences does not need to be processed by "$s2$"(lines 13-14 in Fig. 2); and decides task parallelism and placement based on resource availability. ES loads corresponding operations into a task and starts its execution. The above process repeats.

Programming model: F^2 extends the programming model of existing frameworks [23, 40, 14, 35] in two key ways - (1) user no longer provides low-level details (e.g., stage parallelism); and (2) user can write custom modules to monitor data (Fig. 2 lines 4 – 6) and use them to alter stage computation on-the-fly (Fig. 2 lines 7 – 17).

Example: Consider a simple batch analytics program (Fig. 2) that prints words with more than 100 occurrences. The programmer first provides this to the F^2 client (Fig. 3). The ES starts executing the root logical vertex ("$s1$"), and pushes its output to the datastore (steps ❶-❸). The remaining steps are outlined in Fig. 3's caption, and covered in greater detail later. The same control flow is adopted by graph as well as streaming jobs.

We provide details of the API and F^2's algorithmic and design innovations in the next few sections.

4 Data Store

All jobs' intermediate data is written to/read from a separate data store. Similar to today, we structure data as <key,value> pairs. In batch/stream analytics, the keys are generated by the stage computation logic itself; in graph analytics, keys are identifiers of vertices to which messages (values) are destined for processing in the next iteration. We now address how this data is organized in the

store. Naively writing <k,v> pairs to random machines impacts consumer tasks' read performance. Writing all data corresponding to a key range, irrespective of which producer task generated it, into one file on a randomly-chosen machine ensures consumer task data locality. But such a task-centric view cannot support cross-job isolation; also, "popular" <k,v> pairs can result in data skew, and create I/O hotspots at machines hosting them.

An ideal data organization should achieve three *mutually-conflicting* goals: (1) it should *load balance* data across machines, specifically, *avoid hotspots, improve isolation*, and *minimize skew* in data processing. (2) It should maximize *data locality* by co-locating as much data having the same key as possible. (3) It should be *fault tolerant* - when a storage node fails, recovery should have minimal impact on job runtime.

Next, we describe data storage granularity, which forms the basis for our first-order goals, load balance and low skew. We then show how we meet other objectives.

4.1 Granule: A Unit of Data in F^2

F^2 groups intermediate data based on <key>s. Our grouping is based on an abstraction called granules. Each stage's intermediate data is organized into N granules; N is a large system-wide constant. Each granule stores all <k, v> data from a *fixed, small* key range (total key range split N-ways). F^2 strives to materialize all granule data on one machine; whereas in today's systems, data from the same key range may be written at different producers' locations. This materialization property of granules forms the basis for consumer task data locality. A granule may be spread across machines in the low-likelihood event that the runtime data size corresponding to its key range is unexpectedly large (> a threshold).

Note that key ranges of the intermediate data partitions of today's systems are tied to pre-determined parallelism of consumer tasks, whereas granule key ranges are unrelated to compute structure; they are also generally much smaller. The small size and compute-structure agnosticity help both in managing skew and determining runtime parallelism. The analogy is with bin-packing: for the same total size (amount of data), smaller individual balls (granules) can be better packed (efficiently, and with roughly equal final load) into the fewest bins (compute units).

4.2 Allocating Granules to Machines

We consider how to place multiple jobs' granules to avoid hotspots, reduce per-granule spread (for data locality) and minimize job runtime impact on data loss. We formulate a binary integer linear program (see Fig. 4) to this end. The indicator decision variables, x_i^k, denote that all future data to granule g_k is materialized at machine M_i. The ILP

Objectives (to be minimized):

O_1	$\max_i \left(\sum_k (b_i^k + x_i^k e^k) \right)$
O_2	$\sum_k \left(P^k - \left(\sum_{i \in I_-^k} x_i^k \right) b_{i(k)}^k + \sum_{i \in I_+^k} x_i^k \left(b_i^k + e^k \right) \right)$
O_3	$\sum_k \left((1 - f^k) \sum_{i \in I^\circ} x_i^k \right)$

Constraints:

C_1	$\sum_{k:J(g_k)=j} \left(b_i^k + x_i^k e^k \right) \leq Q_j, \quad \forall j, i$

Variables:

x_i^k	Binary indicator denoting granule g_k is placed on machine i

Parameters:

b_i^k	Existing number of bytes of granule g_k in machine i
e^k	Expected number of remaining bytes for granule, g_k
P^k	$e^k + \sum_i b_i^k$
$J(g_k)$	The job ID for job g_k
$\tilde{i}(k)$	$\text{argmax}_i \quad b_i^k$
I_-^k, I_+^k	$\{i : b_i^k \leq b_{i(k)}^k - e^k\}, \{i : b_i^k > b_{i(k)}^k - e^k\}$
f^k	Binary parameter indicating that granules for same stage as g_k share locations with granules for preceding stages
I°	Set of machines where granules of preceding stages are stored
Q_j	Administrative storage quota for job, j.

Figure 4: Binary ILP formulation for granule placement.

finds the best x_i^k's that minimizes a multi-part weighted objective function, one part each for the three objectives mentioned above.

The first part (O_1) represents the maximum amount of data stored across all machines across all granules. Minimizing this ensures load balance and avoids hotspots. The second part (O_2) represents the sum of *data-spread penalty* across all granules. Here, for each granule, we define the primary location as the machine with the largest volume of data for that granule. The total volume of data in non-primary locations is the data-spread penalty, incurred from shuffling the data prior to processing it. The third part (O_3) is the sum of fault-tolerance penalties across granules. Say a machine m storing intermediate for current stage s fails; then we have to re-execute s to regenerate the data. If the machine also holds data for ancestor stages of s then multiple stages have to be re-executed. If we ensure that data from parent and child stages are stored on different machines, then, upon child data failure only the child stage has to be executed. We model this by imposing a penalty whenever a granule in the current stage is materialized on the same machine as the parent stage. Penalties O_2, O_3 need to be minimized.

Finally, we impose isolation constraint (C_1) requiring the total data for a job to not exceed an administrator set quota Q_j. Quotas help ensure isolation across jobs.

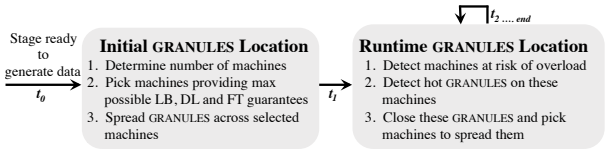


Figure 5: Overview of data organization in F^2.

h1	*// Q_j: max storage quota per job j and machine m.* Based on fairness considerations across all runnable jobs J.
h2	*// M_v: number machines (out of M) to organize data that will be //generated by v of j.* a. Count number machines M_{j75} where j is using $< 75\%$ of Q_j; b. $M_v = max(2, M_{j75} \times \frac{M - M_{j75}}{M})$.
h3	*// Given M_v, compute list of machines $\overrightarrow{M_v}$.* Considers only machines where j is using $< 75\%$ of Q_j; a. Pick machines that provide LB[5], DL[6] and maximum possible FT[7]; b. If $\lvert\overrightarrow{M_v}\rvert < M_v$, relax FT guarantees and pick machines that provide LB and DL; c. If $\lvert\overrightarrow{M_v}\rvert$ is still $< M_v$, pick machines that just provide LB.
h4	*// How to spread granules across $\overrightarrow{M_v}$?* Uniformly spread: $\frac{\lvert\lvert granules \rvert\rvert}{M_v}$ per machine.
h5	*// Which machines are at risk of violating Q_j?* $\overrightarrow{M_j}$: machines which store data of j and j is using $\geq 75\%$ of Q_j.
h6	*// Which granules are hot on $\overrightarrow{M_j}$?* Significantly larger in size or have a higher increasing rate than others.

Table 1: Heuristics employed in data organization

4.3 Fast Granule Allocation

Solving the above ILP at scale can take several seconds delaying granule placement. Further, since the ILP considers granules from multiple stages across jobs, stages' data needs to be batched first, which adds further delay.

F^2 instead uses a simpler, practical approach for the granule placement problem. First, instead of jointly optimizing global placement decisions for all the granules, F^2 solves a "local" problem of placing granules for each stage independently; when new stages arrive, or when existing granules may exceed job quota on a machine, new locations for some of these granules are determined. Second, instead of solving a multi-objective optimization, F^2 uses a linear-time rule-based heuristic to place granules; the heuristic prioritizes load and locality (in that order) in case machines satisfying all objectives cannot be found. Isolation (quota) is always enforced.

Granule location for new stages: (Fig.5) When a job j is ready to run, DS invokes an admin-provided heuristic h1 (Table 1) that assigns j a quota Q_j per machine.

When a stage v of job j starts to generate intermediate data, DS invokes h2 to determine the number of machines M_v for organizing v's data. h2 picks M_v between 2 and a fraction of the total machines which are below 75% of the quota Q_j for j. $M_v \geq 2$ ensures opportunities for data parallel processing (§6.1); a bounded M_v (Table 1) controls shuffle cost when data is processed (§6.1).

Given M_v, DS invokes h3 to generate a list of machines $\overrightarrow{M_v}$ to materialize data on. It starts by creating three sub-lists: (1) For load balancing, machines are sorted lightest-load-first, and only ones which are $\leq 75\%$ quota usage for the corresponding job are considered. (2) For data locality, we prefer machines which already materialize other granules for this stage v, or granules from other stages whose output will be consumed by same downstream stage as v (e.g., the two map stages in Fig. 1a). (3) For fault tolerance, we pick machines where there are no granules from any of v's k upstream stages in the job, sorted in descending order of k.[8] From the sub-lists, we pick least loaded machines that are data local and provide as *high fault tolerance as possible*.

If despite completely trading off fault tolerance – i.e., reaching the bottom of the fault tolerance sub-list – the number of machines picked falls below M_v, we trade-off data locality as well and simply pick least-loaded machines. Finally, given $\overrightarrow{M_v}$, DS invokes h4 and uniformly spreads the granules across the machines.

New locations for existing granules: (Fig.5) Data generation patterns can significantly vary across different stages, and jobs, due to heterogeneous compute logics, data skew, etc. Thus a job j may run out of its quota Q_j on machine m, leaving no room to grow already-materialized granules of j on m. Thus, DS periodically reacts to runtime changes by determining for every j: (1) which machines are at risk of being overloaded; (2) which granules on these machines to spread at other locations; and (3) on which machines to them spread to.

Given a job j, DS invokes h5 to determine machines where j is using at least 75% of its quota Q_j. DS then starts *closing* some granules of j on these machines; future intermediate data for these is materialized on another machine, thereby mitigating any potential hotspot. Specifically, DS invokes h6 to pick granules that are either significantly larger in size or have a higher size increase rate than others for j on m. These granules are more likely to dominate the load and potentially violate Q_j. Focusing on them bounds the number of granules that will spread out. DS groups the granules selected based on the stage which generated them, and invokes heuristic h3 as before to compute the set of machines where to spread. Grouping helps to maximize data locality, and using h3 provides load balance and fault tolerance.

5 Data Visibility

A separate store for intermediate data, organized as above, enables run-time programmable data monitoring. This then supports data-driven computation, as discussed next.

[8] Thus, for the largest value of k, we have all machines that do not store data from *any* of v's ancestors; for $k = 1$ we have nodes that store data from the immediate parent of v

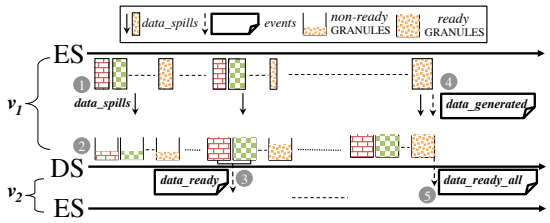


Figure 6: Events and how they facilitate data-compute interaction

5.1 Data Monitoring

F^2 consolidates a granule in a single file, materialized at one or a few locations (§4.1). Thus, granules can be analyzed in isolation, simplifying data visibility.

F^2 supports both *built-in* and *customizable* modules that periodically gather statistics per granule, spanning properties of keys and values. These statistics are carried to a per-stage master (DS-SM, described below) where they are aggregated before being used by ES to take further data-driven actions (§5.2). In parallel, *data* events from the DS-SM carry granule statistics to the F^2 client to aid with data-driven compute logic changes (§6.3).

Built-in modules run constantly and collect statistics such as current granule size, number of (k,v) pairs and rate of growth; in addition to supporting applications, these are used by the store in runtime data organization (§4.3).

Custom modules are programmer-provided UDFs (user defined functions). Since supporting arbitrary UDFs can impose high overhead, we restrict UDFs to those that can execute in linear time and O(1) state. We provide a library of common UDFs, such as computing the number of entries for which values are $<, =$, or $>$ than a threshold.

The distributed datastore with small granules, and distributed DS-SMs, ensure monitoring is naturally *scalable*.

5.2 Acting on Monitored Data Properties

The ability to provide data visibility at runtime, along with a decoupled compute and data architecture, enables F^2 to cleanly support data-driven computation. At a high level, DS triggers computation when certain data properties are met through an *event* abstraction, and the ES performs corresponding computation on the given data.

F^2 introduces the events shown in Fig. 6. ❶ When a stage v_1 generates a batch of intermediate data, a $data\_spill$ containing the data is sent to the DS, ❷ which accumulates it into granules (§4). The DS-SM is made aware of the number of v_1 granules. ❸ Whenever the DS-SM determines that a collection of v_1's granules are ready for further processing, it sends a $data\_ready$ event per granule to the ES for subsequent processing by tasks of a consumer stage v_2. This event carries per-granule information such as: a list of machine(s) on which each granule is spread, and a list of (aggregated) statistics (counters) collected by the *built-in* data modules. ❹ Finally, a $data\_generated$ event (from ES) notifies the v_1 DS-SM that v_1 finished generating $data\_spills$ (due to v_1 computation completing). ❺ Subsequently, DS-SM notifies the ES through a $data\_ready\_all$ event that all the data generated by v_1 is ready for consumption (i.e., $data\_ready$ events were sent by the DS-SM for all granules of v_1). The pair of $<data\_generated, data\_ready\_all>$ events thus enables ES to determine when an immediate downstream stage v_2, that is reading the data generated by v_1, has received all of its input data (Fig. 6).

The key enabler of this interaction is a *ready trigger* that enables the per-stage DS-SM to decide when (a collection of) granules can be deemed ready for corresponding computation. The trigger logic is based on statistics collected by the data modules. An F^2 program can provide *custom ready triggers* for each of its stages, which the F^2 client transfers to the DS-SMs. F^2 implements a *default ready trigger* which is otherwise applied.

Default ready trigger: Here, the DS-SM deems granules ready when the computation generating them is done; this is akin to a barrier in existing batch analytics and bulk synchronous execution in graph analytics. For a streaming job, the DS-SM deems a granule ready when it has $\geq X$ records (X is a system parameter that a user can configure) from producer tasks, and sends a $data\_ready$ event to ES. On receiving this, ES executes a consumer stage task on this granule. This is akin to micro-batching in existing streaming systems [51], with the crucial difference that the micro-batch is not wall clock time-based, but is based on the more natural intermediate data count.

Custom ready trigger: If available, the DS-SM deems granules as ready using custom triggers. Programmers define these triggers based on knowledge about the semantics of the computation performed, and the type of data properties sought.

E.g., consider the partial execution of a batch (graph) analytics job, consisting of the first two logical stages (first two iterations) $v_1 \rightarrow v_2$. If the processing logic in v_2 contains commutative+associative operations (e.g., sum, min, max, count, etc..), it can start processing its input before all of it is in place. For this, the user can define a *pipelining custom ready trigger*, and instruct DS-SM to consider a granule generated by v_1 ready whenever the number of records in it reaches a threshold X. This enables ES to overlap computation, i.e., execute v_2 as follows: upon receiving a $data\_ready$ event from the DS-SM, it launches tasks of v_2; tasks read the current data, compute the associative+commutative function on the (k,v) data read, and push the result back to DS (in the same granules advertised through the received

$data\_ready$ event; note that the key remains the same). The DS-SM waits for each granule to grow back beyond threshold X for generating subsequent $data\_ready$ events. Finally, when a $data\_generated$ event is received from v_1, the DS-SM triggers a final $data\_ready$ event for all the granules generated by v_1, and a subsequent $data\_ready\_all$ event, to enable v_2's final output to be written in granules and fully consumed by a downstream stage, say v_3 (similar to Fig. 6). Such pipelining speeds up jobs in *batch* and *graph* analytics (see. §8.1.2).

DS support for custom data modules enables definitions of even broader custom triggers. E.g., in case of a streaming job, v_2 may (re)compute a weighted moving average whenever 100 data points with distinct keys are generated by v_1. Here, the user would write a custom monitoring module to enable the DS to collect statistics regarding entries with distinct keys (across granules), and a custom ready trigger executed at the DS-SM to mark all granules of v_1 as ready whenever 100 new entries with distinct keys were generated across all of them. Such *data-driven stream analytics* performs much faster and is more efficient than today's stream systems (§8.1.3).

6 Execution Service

The ES manages computation: given intermediate data and available resources, it determines optimal parallelism and deploys tasks to minimize skew and shuffle, and it maps granules to tasks in a resource-aware fashion. The ES design naturally mitigates stragglers, and facilitates data-driven compute logic changes.

6.1 Task Parallelism, Placement, and Sizing

Given a set of ready granules (G) for a stage, V, the ES maps subsets of granules to tasks, and determines the location (across machines M) and the size of the corresponding tasks based on available resources. This multi-decision problem, which evens out data volume processed by tasks in a stage, and minimizes shuffle subject to resource contraints on each machine, can be cast as a binary ILP (omitted for brevity). However, the formulation is non-linear; even a linear version (with many variables) is slow to solve at scale. For tractability, we propose an *iterative procedure* (below) that applies a set of heuristics (Tab. 2) repeatedly until tasks for all ready granules are allocated, and their locations and sizes determined.

In each iteration, we first group granules, G, into a collection of subsets, $\overrightarrow{G}$, using h7; the number of subsets, $|\overrightarrow{G}|$, is determined by the size of the largest granule (see line (a)). Grouping minimizes data spread within each subset (line (b)), and spreads total data evenly across subsets (line (c)) making cross-task performance uniform.

h7	*// $\overrightarrow{G}$: subsets of unprocessed granules.* a. $GrMax = 2 \times \lvert g \rvert$, g is largest granule $\in G$; b. Group all granules $\in G$ into subsets in strict order: i. data local granules together; ii. each spread granule, along data-local granules together; iii. any remaining granules together; subject to: iv. each subset size $\leq GrMax$; v. conflicting granules don't group together; vi. troublesome granules always group together.
h8	*// $\overrightarrow{M}$: preferred machines to process each subset $\in \overrightarrow{G}$.* c. no machine preference for troublesome subsets $\in \overrightarrow{G}$ d. for every other subset $\in \overrightarrow{G}$ pick machine m such that: i. all granules in the subset are only materialized at m; ii. otherwise m contains the largest materialization of the subset.
h9	Compute $\overrightarrow{R}$: resources needed to execute each subset $\in \overrightarrow{G}$: e. $\overrightarrow{A}$ = available resources for j on machines $\overrightarrow{M}$; f. $F = \min\left(\frac{\overrightarrow{A}[m]}{\text{total size of granules allocated to } m}, \text{for all } m \in \overrightarrow{M}\right)$; g. for each subset $i \in \overrightarrow{G}$: $\overrightarrow{R}[i] = F \times$ total size of granules allocated to $\overrightarrow{G}[i]$.

Table 2: Heuristics to group granules and assign them to tasks

Then, we determine a preferred machine to process each subset; this is a machine where most if not all granules in the subset are materialized (h8). Choosing a preferred machine in this manner may cause starvation if the subset of granules cannot be processed due to resource unavailability at the machine. For the rest of the iteration, we ignore granules in such subsets and re-consider them with a different grouping in later iterations (line b.v, h7).

Next, we assign a task for each subset of granules which can be processed, and allocate resources *altruistically* to the tasks using heuristic h9. Given available resources across machines, $\overrightarrow{A}$, we first compute the minimum resource available to process unit data (F; line (f)). Then, for each task, we assign $F \times |\overrightarrow{G}[i]|$ resources (line (f)), i.e., the resource allocated is F times the total data in the subset of granules allocated to the task.

Allocating resources proportional to input size coupled with roughly equal group sizes, ensures that tasks have roughly equal finish times in processing their subsets of granules. Furthermore, by allocating resources corresponding to the minimum available, our approach realizes *altruism*: if a job gets more resources than what is available for the most constrained group, then it does not help the job's completion time (because completion time depends on how fast the most constrained group is processed). Altruistically "giving back" such resources helps speed up other jobs or other stages in the same job [27].

The above steps repeat whenever new granules are ready, or existing ones can't be scheduled. Similar to [49], we attempt several tries to execute a group, before regrouping conflicting groups. Finally, if some granules cannot be executed under any grouping, we mark them as troublesome and process them on any machine (line (c)).

6.2 Handling Task Failures and Stragglers

Stragglers: Data organization into granules and late-binding computation to data enables a natural, simple, and effective straggler mitigation technique. If a task struck by resource contention makes slower progress than others in a stage, then the ES simply splits the task's granule group into two, in proportion of the task's processing speed relative to average speed of other tasks in the stage. It then assigns the larger-group granules to a new task, and places it using the approach above. Thus, F^2 addresses stragglers via natural work reallocation, as opposed to using clone tasks [52, 17, 15, 34, 16] in compute-centric frameworks which waste resources and duplicates work.

Task failures: Similar to existing frameworks, when an F^2 task fails, only the failed tasks need to be re-executed. However, this will result in duplicates data in all granules for the stage leading to intermediate data inconsistencies. To avoid duplicates, we use checksums at the consumer task-side F^2 library to suppress duplicate spills.

6.3 Runtime DAG Changes

Events and visibility into data enable run-time logic changes in F^2. We introduce *status* events to help the ES query the F^2 client to check if the user program requires alternate logic to be launched based on observed statistics.

Upon receiving *data_ready* events from the DS-SM, ES sends *status* events to the F^2 client to interrogate regarding how to process each granule it received notification for. Given the user-provided compute logic and granule statistics obtained through *data_ready* events, the F^2 client then notifies the ES to take one the following actions: (1) *no new action* – assign computation as planned; (2) *ignore* – don't perform any computation[9]; (3) *replace computation* with new logic supplied by the user. When launching a task, the ES provides it with the updated computation logic for each granule the task processes.

7 Implementation

We prototyped F^2 by modifying Tez [5] and leveraging YARN [45]. F^2's core components are application agnostic and support diverse analytics as shown in §8.

The DS was implemented from scratch and consists of three kinds of daemons - *Data Service Workers* (DS-W), *Data Service Master* (DS-M) and *Data Service Stage Masters* (DS-SM, §5.2). We leverage YARN to launch/terminate them. The DS-M is a cluster-wide daemon that manages data organization across DS-Ws, and manages DS-SMs. The per-stage DS-SM collects and stores granule statistics from DS-Ws. The DS-W runs on cluster machines and conducts node-level data management. DS-W handles storing the data it receives from the ES or from other DS-Ws in a local in-memory file system (`tmpfs` [42]) and transfers data to other DS-Ws per DS-M directives. It collects statistics and reports to the DS-M and DS-SMs via heartbeats. Finally, it provides ACKs to ES-tasks (described below) for the data they are pushing.

[9]The user program may deem an entire granule to not have any useful data to compute on

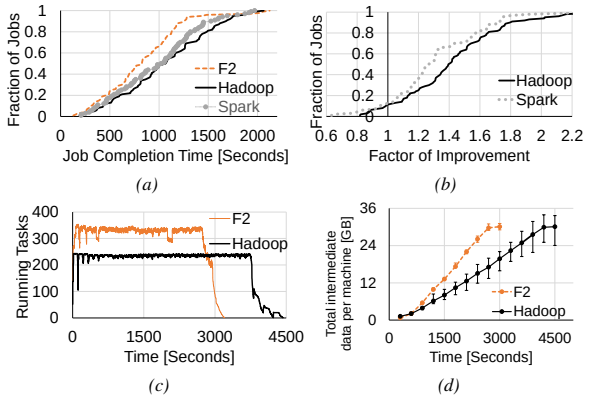


Figure 7: (a) CDF of JCT; (b) CDF of factors of improvement of individual jobs using F^2 w.r.t. baselines [2, 50]; (c) Running tasks; (d) Per machine average, min *and* max *storage load.*

The ES was implemented by modifying components of Tez to enable data-driven execution. It consists of a single ES-Master (ES-M) responsible for launching ES-tasks and determining parallelism, placement, and sizing. It receives $data\_ready$ events from DS-SMs and is responsible for making run time logic changes via $status$ events. ES-tasks are modified Tez tasks that have an interface to the local DS-W as opposed to local disk or cluster-wide storage. The F^2 client is a standalone process per-job; it interacts with the DS-SM and ES-M as shown in Fig. 3.

All communication (asynchronous) between DS, ES and F^2 client is implemented through RPCs [12] in YARN using Google Protobuf [11]. We also enhanced the RPC protocol to enable communication between the YARN Resource Manager (RM) and ES-M to propagate resource allocation for a job as computed in the RM.

8 Evaluation

We evaluated F^2 on a 50-machine cluster deployed on the Utah CloudLab [7] using publicly available benchmarks – *batch* TPC-DS jobs, PageRank for *graph analytics*, and synthetic *streaming* jobs. We set F^2 to use default ready triggers, and equal storage quota ($Q_j = 2.5GB$).

8.1 F^2 in Testbed Experiments

Workloads: We consider a mix of jobs, all from TPC-DS (**batch**), or all from PageRank (**graph**). In each experiment, jobs are randomly chosen and follow a Poisson arrival distribution with average inter-arrival time of 20s.

Each job lasts up to 10s of minutes, and takes as input tens of GBs of data. Since jobs arrive and finish arbitrarily, the resource availability during the course of a job's execution fluctuates. For **streaming**, we created a synthetic workload from a job which periodically replays GBs of text data from HDFS and returns top 5 most common words for the first 100 distinct words found. We run each experiment thrice and present the median.

Cluster, baseline, metrics: Our machines have 8 cores, 64GB of memory, 256GB storage, and a 10Gbps NIC. The cluster network has a congestion-free core. We compare F^2 as follows: (1) **Batch frameworks**: *vs.* Tez [5] running atop YARN [45], for which we use the shorthand "Hadoop" or "CC"; and *vs.* SparkSQL [18] atop Spark; (2) **Graph processing**: *vs.* Giraph (i.e., open source Pregel [35]); (3) **Streaming**: *vs.* SparkStreaming [51]. We study the relative improvement in the average job completion time (JCT), or $\text{Duration}_{CC}/\text{Duration}_{F^2}$. We measure efficiency using makespan.

8.1.1 Batch Analytics

Performance and efficiency: Fig. 7a shows the JCT distributions of F^2, Hadoop, and Spark for the TPC-DS workload. Only 0.4 (1.2) highest percentile jobs are worse off by $\leq 1.06\times$ ($\leq 1.03\times$) than Hadoop (Spark). F^2 speeds up jobs by $1.4\times$ ($1.27\times$) on average, and $2.02\times$ ($1.75\times$) at $95th$ percentile w.r.t. Hadoop (Spark). Also, F^2 improves makespan by $1.32\times$ ($1.2\times$).

Fig. 7b presents improvement for individual jobs. For more than 88% of the jobs, F^2 outperforms Hadoop and Spark. Only 12% jobs slow down to $\leq 0.81\times$ ($0.63\times$) using F^2. Gains are $> 1.5\times$ for $> 35\%$ of the jobs.

Sources of improvements: We now dig in to understand the benefits. We observe *more rapid processing*, and *better data management* are key underlying causes.

First, we snapshot the number of running tasks across all the jobs in one of our experiments when running F^2 and Hadoop (Fig. 7c). F^2 has $1.45\times$ more tasks scheduled over time which directly translates to jobs finishing $1.37\times$ faster. It has $1.38\times$ better cluster efficiency than Hadoop. Similar observations hold for Spark (omitted).

The main reasons for rapid processing/high efficiency are two-fold. (1) DS's data organization (§4.3) naturally ensures that most tasks are data local (76% in our expts). This improves average *consumer* task completion time by $1.59\times$. Resources thus freed can be used by other jobs' tasks. (2) Our ES can provide similar input sizes for tasks in a stage (§6.1) – within 14.4% of the mean input size per task. Thus, we see predictable per-stage performance and better resource utilization (more in §8.3).

Second, Fig. 7d shows the size of the cross-job total intermediate data per machine. We see that Hadoop generates heavily imbalanced load spread across machines. This creates many storage hotspots and slows down tasks competing on those machines. Spark is similar. F^2 avoids hotspots (§4.2) improving overall performance.

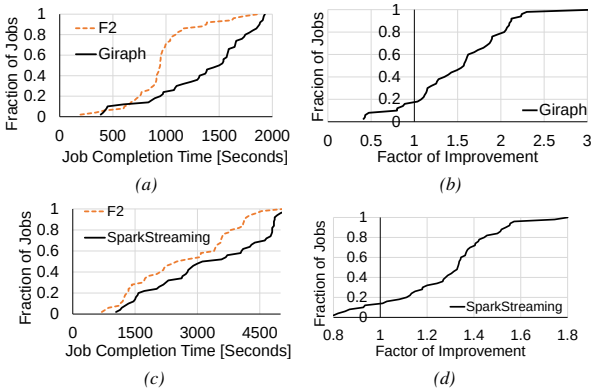


Figure 8: (a) CDF of JCT using F^2 and Giraph [1]; (b) CDF of factors of improvement of individual jobs using F^2 w.r.t. Giraph; (c) CDF of JCT using F^2 and SparkStreaming [51]; (d) CDF of factors of improvement of individual jobs using F^2 w.r.t. SparkStreaming.

F^2 slowdown: We observe jobs generating less intermediate data are more prone to performance losses, especially under ample resource availability. One reason is that F^2 strives for granule-local task execution (§6.1). However, if resources are unavailable, F^2 will assign the task to a data-remote node, or get penalized waiting for a data-local placement. Also, F^2 gains are lower w.r.t. Spark. This is mainly an artifact of our implementation atop Hadoop, and a non-optimized in-memory store.

We found that only 18% of granules across all jobs are spread across machines. $> 25\%$ of the jobs whose performance improves processed "spread-out" granules. Only $\leq 14\%$ of the slowed jobs processed spread granules.

8.1.2 Graph Processing

We run multiple PageRank (40 iters.) jobs on the Twitter Graph [21, 20]. F^2 groups data (messages exchanged over algorithm iterations) into granules based on vertex ID. We use a *custom ready trigger* (§5.2) so that a granule is processed only when ≥ 1000 entries are present.

Fig. 8a shows the JCT distribution of F^2 and Giraph. Only 0.6%-ile of jobs are worse off by $\leq 1.35\times$ than Giraph. F^2 speeds up jobs by $1.57\times$ on average and $2.24\times$ at the $95th$ percentile (Fig. 8b). Only 16% of the jobs slow down. Gains are $> 2\times$ for $> 20\%$ of the jobs.

Improvements arise for two reasons. First, F^2 is able to *deploy appropriate number of tasks only when needed*: custom ready triggers immediately indicate data availability, and runtime parallelism (§6.1) allows messages to dense vertices [26] to be processed by more than one task. In our experiments, F^2 has $1.53\times$ more tasks (each runs

multiple vertex programs) scheduled over time; rapid processing and runtime adaptation to data directly translates to jobs finishing faster. Second, because of triggered compute, F^2 does not necessarily hold resources for a task if not needed, resulting in $1.25\times$ better cluster efficiency.

As before, ES stickiness to achieve data-locality can slow down completion times in some cases.

8.1.3 Stream Processing

Here, we configure the Spark Streaming batch interval to 1 minute; the F^2 ES repeats job logic execution at the same time interval. Also, we implemented a *custom ready trigger* to enable computation whenever ≥ 100 distinct entries are present in intermediate data. Figures 8c, 8d show our results. F^2 speeds up jobs by $1.33\times$ on average and $1.55\times$ at the $95th$ %-ile. Also, 15% of the jobs are slowed down to around $0.8\times$.

The main reason for gains is F^2's *ability to trigger computation based on data properties through custom ready triggers*; F^2 does not have to delay execution till the next time interval if data can be processed now. A SparkStreaming task has to wait as it has no data visibility. In our experiments, more than 73% of the executions happens at less than $40s$ time intervals with F^2.

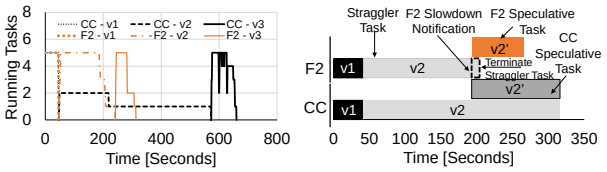
F^2 suffers somewhat due to an implementation artifact atop a non-optimized stack for streaming: launching tasks in YARN is significantly slower than acquiring tasks in Spark. This can be exacerbated by ES stickiness. However, such effects are mitigated over long job run times.

8.1.4 F^2 Overheads

CPU, memory overhead: We find that DS-W (§7) processes inflate the memory and CPU usage by a negligible amount even when managing data in close to storage capacity. DS-M and DS-SM have similar resource profiles.
Latency overhead: DS-M interacts with various system components. We compute the average time to process heartbeats from ES, DS-SM, DS-W and F^2 client. For 5000 heartbeats, the time to process each is $2-5ms$. We implemented the F^2 client and ES logic atop Tez AM. Our changes inflate AM decision logic by $\leq 14ms$ per request with a negligible increase in AM memory/CPU.
Network overhead from events/heartbeats is negligible.

8.2 Contention and Isolation

We vary storage load, and hence resource contention, by changing the number of machines while keeping the workload constant; half as many servers lead to twice as much load. We see that at $1\times$ cluster load, F^2 improves over CC by $1.39\times$ ($1.32\times$) on average in terms of JCT (makespan). Even at high contention (up to $4\times$), F^2's gains keep increasing ($1.83\times$ and $1.42\times$, resply.). This



(a) *(b)*

Figure 9: (a) Controlling task parallelism significantly improves F^2's performance over CC. (b) Straggler mitigation with F^2 and CC.

is expected because: (1) F^2 *minimizes resource wastage* and the time spent in shuffling the data, due to its ability to execute on local, ready data; (2) data is load balanced leading to *few storage hotspots*, and *isolation is better*.

8.3 Data-driven Computation Benefits

The overall benefits of F^2 above also included the effects of dynamic parallelism/placement/sizing (§6.1) and straggler mitigation (§6.2). We did not fully delineate these effects to simplify explanation. We now delve deeper into them to shed more light on data-driven computation benefits. We run microbenchmarks on a 5 machine cluster and a mix of toy jobs $\mathbb{J}_1$ ($v_1 \rightarrow v_2$) and $\mathbb{J}_2$ ($v_1 \rightarrow v_2 \rightarrow v_3$).
Skew and parallelism: Fig. 9a shows the execution of one of the $\mathbb{J}_2$ jobs from our workload when running F^2 and CC. F^2 improves JCT by $2.67\times$ over CC. CC decides stage parallelism tied to the number of data partitions. That means stage v_1 generates 2 intermediate partitions as configured by the user and 2 tasks of v_2 will process them. However, execution of v_1 leads to data skew among the 2 partitions ($1GB$ and $4GB$).On the other hand, F^2 partitions intermediate data in granules of $0.5GB$ each and decides at runtime a max. input size per task of $1GB$. This leads to running 5 tasks of v_2 with equal input size and $2.1\times$ faster completion time of v_2 than CC.

Over-parallelizing execution does not help. With CC, v_2 generates 12 partitions processed by 12 v_3 tasks. Under resource crunch, tasks get scheduled in multiple waves (at 570s in Fig. 9a) and completion time for v_3 suffers (85s). In contrast, F^2 assigns at runtime only 5 tasks of v_3 which can run in a single wave; v_3 finishes $1.23\times$ faster.
Straggler mitigation: We run an instance of $\mathbb{J}_1$ with 1 task of v_1 and 1 task of v_2 with an input size of $1GB$. A slowdown happens at the v_2 task. F^2 generates granules of $0.5GB$ each and max. input size is $1GB$. This translates to 2 granules assigned to a v_2 task.

In CC (Fig. 9b), once a straggler is detected (v_2 task at 203s), it is allowed to continue, and a speculative task v_2' is launched that duplicates v_2's work. The work completes when v_2 *or* v_2' finishes (at 326s). In F^2, upon straggler detection, the straggler (v_2) is notified to finish processing the current granule; a task v_2' is launched and assigned data from v_2's unprocessed granule. v_2 finishes

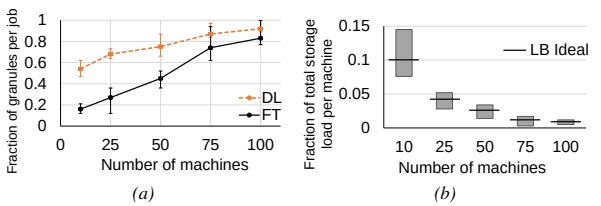


Figure 10: (a) Average, min and max fraction of granules which are data local (DL) respectively fault tolerant (FT) across all the jobs for different cluster load; (b) Max, min and ideal storage load balance (LB) on every machine for different cluster load.

processing the first granule at 202s; v_2' processes the other granule and finishes $1.7\times$ faster than v_2' in CC.

Runtime logic changes: We consider a job which processes words and, for words with < 100 occurrences, sorts them by frequency. The program structure is $v_1 \rightarrow v_2 \rightarrow v_3$, where v_1 processes input words, v_2 computes word occurrences, and v_3 sorts the ones with < 100 occurrences. In CC, v_1 generates $17GB$ of data organized in 17 partitions; v_2 generates $8GB$ organized in 8 partitions. Given this, 17 v_2 tasks and 8 v_3 tasks execute, leading to a CC JCT of 220s. Here, the entire data generated by v_2 has to be analyzed by v_3. In contrast, F^2 registers a custom DS module to monitor #occurrences of all the words in the 8 granules generated by v_2. We implement actions to *ignore* v_2 granules that don't satisfy the processing criteria of v_3 (§6.3). At runtime, 2 data events are triggered by the DS module, and 6 tasks of v_3 (instead of 8) are executed; JCT is 165s ($1.4\times$ better).

8.4 LB vs. DL vs. FT

To evaluate DS load balancing (LB), data locality (DL - each granule consolidated on one machine) and fault tolerance (FT - current stage co-locates no granule with ancestor stages), we stressed the data organization under different cluster load (§4.3). We used job arrivals and all stages' granule sizes from one of our TPC-DS runs.

The main takeaways are (Fig. 10): (1) F^2 prioritizes DL and LB over FT across cluster loads (§4.3); (2) when the available resources are scarce ($5\times$ higher than initial), all three metrics suffer. However, the maximum load imbalance per machine is $< 1.5\times$ than the ideal, while for any job, $\geq 47\%$ of the granules are DL. Also, on average 16% of the granules per job are FT; (3) less cluster load ($0.6\times$ lower than initial) enables more opportunities for DS to maximize all of the objectives: $\geq 84\%$ of the per-job granules are DL, 71% are FT, with at most $1.17\times$ load imbalance per machine than the ideal.

Failures: We study failure impact using 5 machines and a toy $\mathbb{J}_2$ job from above, under two cases: (1) the input for v_3 is on a separate machine m, and (2) input for v_2 and v_3 are located on m. In (1), when only the immediate input (i.e., v_2's output) has to be regenerated, job performance is up to $0.76\times$ worse w.r.t. m not failing. However, in (2) when input dependency chains (i.e., both v_1 and v_2) need regeneration (§4.2), performance degrades significantly ($0.58\times$ for $20GB$ input data).

8.5 Altruism

We also quantified how much F^2's logic to altruistically decide task sizing (§2) impacts job performance. We compare F^2's approach with a greedy task sizing approach, where each task gets $\frac{1}{\#jobs}$ resource share and uses all of it. For the same workload as 8.4, F^2 speeds up jobs by $1.48\times$ on average, and $3.12\times$ ($4.8\times$) at $75th$ percentile ($95th$ percentile) w.r.t. greedy approach. Only 16% of the jobs are slow down by no more than $0.6\times$.

9 Related Work

Decoupling: For batch analytics, PyWren [29] stores intermediate data in an externally managed storage (Amazon S3) and uses stateless functions to read, process, and write data back to S3. Since intermediate data is still opaque, and compute and storage are isolated, PyWren cannot support data-driven computation or achieve data locality. Hurricane [19] decouples to mitigate skew by cloning slow tasks and partitioning work dynamically. However, it also has no data visibility. Its data organization does not consider data locality and fault tolerance.

Naiad [38] and StreamScope [46] also adopt decoupling. They tag intermediate data with vector clocks which are used to trigger compute in the correct order under failures. Thus, both support ordering driven computation, orthogonal to data value-driven computation in F^2. Naiad assumes entire data fits in memory. StreamScope is not applicable to batch/graph analytics.

Skew, logic changes: Many works target data skew in parallel databases [31, 28, 47] and big data systems [34]. F^2 goes beyond them and provides isolation across jobs, supports data-driven computation, mitigates skew, and ensure data locality. Optimus [30] allows changing application logic based on approximate statistics operators that are deployed alongside tasks. However, the system targets just computation rewriting, and cannot enable other data-driven benefits of F^2, e.g., adapting parallelism, straggler mitigation, and scheduling when data is ready.

10 Summary

The compute-centric nature of existing analytics frameworks hurts flexibility, efficiency, isolation, and performance. F^2 reenvisions analytics frameworks, and is inspired by programmable network measurements, serverless platforms, and multi-tenant K-V stores. F^2 cleanly separates computation from intermediate data. Via pro-

grammable monitoring of data properties and a rich event abstraction, F^2 enables data-driven decisions for what computation to launch, where to launch it, and how many parallel instances to use, while ensuring isolation. Our evaluation using batch, stream and graph workloads shows that F^2 outperforms state-of-the-art frameworks.

References

[1] Apache Giraph. http://giraph.apache.org.

[2] Apache Hadoop. http://hadoop.apache.org.

[3] Apache Hive. http://hive.apache.org.

[4] Apache Samza. http://samza.apache.org.

[5] Apache Tez. http://tez.apache.org.

[6] AWS Lambda. https://aws.amazon.com/lambda/.

[7] Cloudlab. https://cloudlab.us.

[8] Hadoop Capacity Scheduler. https://bit.ly/2HKKbMm.

[9] Multi-Tenant Storage with Amazon DynamoDB. https://amzn.to/2jsCXOH.

[10] Presto — Distributed SQL Query Engine for Big Data. prestodb.io.

[11] Protocol Buffers. https://bit.ly/1mISy49.

[12] Remote Procedure Call. https://bit.ly/2rjaUVo.

[13] Spark SQL. https://spark.apache.org/sql.

[14] Storm: Distributed and fault-tolerant realtime computation. http://storm-project.net.

[15] S. Agarwal, S. Kandula, N. Burno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing data parallel computing. In *NSDI*, 2012.

[16] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In *NSDI*, 2013.

[17] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in mapreduce clusters using Mantri. In *OSDI*, 2010.

[18] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in Spark. In *SIGMOD*, 2015.

[19] L. Bindschaedler, J. Malicevic, N. Schiper, A. Goel, and W. Zwaenepoel. Rock you like a hurricane: Taming skew in large scale analytics. In *EuroSys*, 2018.

[20] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *WWW*, 2011.

[21] P. Boldi and S. Vigna. The webgraph framework i: Compression techniques. In *WWW*, 2004.

[22] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with Orchestra. In *SIGCOMM*, 2011.

[23] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[24] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[25] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[26] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[27] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *OSDI*, 2016.

[28] K. A. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. In *VLDB*, 1991.

[29] E. Jonas, Q. Pu, S. Venkataraman, I. Stoice, and B. Recht. Occupy the cloud: Distributed computing for the 99%. In *SOCC*, 2017.

[30] Q. Ke, M. Isard, and Y. Yu. Optimus: A dynamic rewriting framework for data-parallel execution plans. In *EuroSys*, 2013.

[31] M. Kitsuregawa and Y. Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (sdc). In *VLDB*, 1990.

[32] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*, 2015.

[33] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *SIGMOD*, 2015.

[34] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: Mitigating skew in mapreduce applications. In *SIGMOD*, 2012.

[35] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.

[36] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib: Machine learning in Apache Spark. *CoRR*, abs/1505.06807, 2015.

[37] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Trumpet: Timely and precise triggers in data centers. In *SIGCOMM*, 2016.

[38] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP*, 2013.

[39] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: A Universal Execution Engine for Distributed Data-Flow Computing. In *NSDI*, 2011.

[40] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *SIGMOD*, 2015.

[41] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *OSDI*, 2012.

[42] P. Snyder. tmpfs: A virtual memory file system. In *European UNIX Users' Group Conference*, 1990.

[43] A. Thusoo, R. Murthy, J. S. Sarma, Z. Shao, N. Jain, P. Chakka, S. Anthony, H. Liu, and N. Zhang. Hive – a petabyte scale data warehouse using Hadoop. In *ICDE*, 2010.

[44] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *SIGMOD*, 2014.

[45] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *SoCC*, 2013.

[46] L. Wei, Q. Zhengping, X. Junwei, Y. Sen, Z. Jingren, and Z. Lidong. Streamscope: Continuous reliable distributed processing of big data streams. In *NSDI*, 2016.

[47] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen. Handling data skew in parallel joins in shared-nothing systems. In *SIGMOD*, 2008.

[48] L. Yuan, C.-N. Chuah, and P. Mohapatra. Progme: Towards programmable network measurement. *IEEE/ACM Trans. Netw.*, 19(1):115–128, Feb. 2011.

[49] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

[50] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[51] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant stream computation at scale. In *SOSP*, 2013.

[52] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, 2008.