WHIZ: Data-Driven Analytics Execution

Robert Grandl[†*]
Google

Arjun Singhvi[†]
University of Wisconsin–Madison

Raajay Viswanathan[*]
Uber Technologies Inc.

Aditya Akella
University of Wisconsin–Madison

***Abstract*—** Today's data analytics frameworks are compute-centric, with analytics execution almost entirely dependent on the predetermined physical structure of the high-level computation. Relegating intermediate data to a second class entity in this manner hurts flexibility, performance, and efficiency. We present WHIZ, a new analytics execution framework that cleanly separates computation from intermediate data. This enables runtime visibility into intermediate data via programmable monitoring, and data-driven computation where data properties drive when/what computation runs. Experiments with a WHIZ prototype on a 50-node cluster using batch, streaming, and graph analytics workloads show that it improves analytics completion times 1.3-2× and cluster efficiency 1.4× compared to state-of-the-art.

1 Introduction

Many important applications in diverse settings rely on analyzing large datasets, including relational tables, event streams, and graph-structured data. To analyze such data, several execution frameworks have been introduced [4, 7, 15, 24, 36, 42–45, 50, 51]. These enable data parallel computation, where an analytics job's logic is run in parallel on data shards spread across cluster machines.

Almost all these frameworks build on the *MapReduce execution engine* [21]. Like MapReduce, they leverage *compute-centric* execution (§2). Their execution engines' focus is on splitting a job's computational logic, and distributing it across *tasks* to be run in parallel. All aspects of the subsequent execution of the job are rooted in the job's computational logic, and its task-level computation distribution. These include the fact that compute logic running inside tasks is static and/or predetermined; intermediate data is partitioned and routed to where it is consumed based on the task-level structure; and dependent tasks are launched when a fraction of upstream tasks they depend on finish. These attributes of job execution are not related to, or driven by, the properties of intermediate data, i.e., how much and what data is generated. Thus, intermediate data is a *second-class citizen*.

Compute-centricity was a natural early choice: knowing job structure beforehand simplifies carving containers to execute tasks; compute-centricity provided clean mechanisms to recover from failures – only tasks on a failed machine needed to be re-executed; and job scheduling became simple because of having to deal with static inputs, i.e., fixed tasks/dependency structures.

Unfortunately, today, compute-centricity severely hinders analytics performance and cluster efficiency due to four fundamental issues (§2, §9): (1) Intermediate data-unawareness means there is no way to quickly adapt job execution based on changing run-time data properties (e.g., volume, key distribution, etc.) to ensure performance- and resource-optimal data processing. (2) Likewise, static parallelism and intermediate data partitioning inherent to compute-centricity prevent adaptation to intermediate data skew and resource flux which are difficult to predict ahead of time, yet, common to modern datasets [30] and multi-tenancy. (3) Execution schedules being tied to compute structure can lead to resource waste while tasks wait for input data to become available - an effect that is exacerbated under multi-tenancy. (4) The skew due to compute-based organization of intermediate data can result in storage hotspots and poor cross-job I/O isolation; it also curtails data locality.

We observe that the above limitations arise from (1) tight *coupling* between intermediate data and compute, and (2) *intermediate data agnosticity* in today's execution frameworks. To improve analytics performance, efficiency, isolation, and flexibility, we develop a new execution framework, WHIZ, that eschews compute-centricity, cleanly separates computation from all intermediate data, and treats both intermediate data and compute as equal first-class entities during analytics applications' execution. WHIZ applies equally to batch analytics, streaming and graph processing.

In WHIZ, intermediate data is written to/read from a *logically* separate distributed key-value datastore. The store offers programmable visibility – applications can provide custom routines for monitoring runtime data properties. The store

[†]These authors contributed equally to this work.
[*]Work done while at University of Wisconsin–Madison.

notifies an execution layer when an application's runtime data satisfies predicates based on data properties. Decoupling, monitoring, and predicates enable *data-driven incremental computation*: based on data properties, WHIZ decides on the fly *what logic* to launch in order to further process the data generated, *how many parallel tasks* to launch, *when/where to launch* them, and *what resources* to allocate to tasks.

We make the following contributions in designing WHIZ: (1) We present a scalable approach for programmable intermediate data monitoring which forms the basis for data-driven actions. (2) We show how to organize intermediate data from multiple jobs in the datastore so as to achieve data locality, fault tolerance, and cross-job isolation. Since obtaining an optimal data organization is intractable, we develop novel heuristics that carefully trade-off among these objectives. (3) We build an execution layer that incrementally decides all aspects of the job execution based on data property predicates being satisfied. We develop novel iterative heuristics for the execution layer to decide, for each ready-to-run analytics stage, task parallelism, task placement, and task sizing. This minimizes runtime skew in the data processed, lowers data shuffle cost and ensures optimal efficiency under resource dynamics. The execution layer also decides the optimal per-task logic to use at run-time.

We build a WHIZ prototype using Tez [47] and YARN [52] (15K LOC). We conduct experiments on a 50 machine cluster in CloudLab [6]. We compare against several state-of-the-art compute-centric (CC) batch, stream and graph processing approaches. Using data-driven incremental computation, WHIZ improves median (95%-ile) job completion time (JCT) $1.3-1.6\times$ ($1.5-2.2\times$) and cluster efficiency $1.4\times$ by launching the right number of appropriately-sized tasks only when predicates are met. We observe up to $2.8\times$ improvement in efficiency due to WHIZ's ability to change processing logic on the fly. Furthermore, we observe that the impact on JCT under failures is minimal due to WHIZ's data organization. We observe that WHIZ's gains relative to CC improve with contention due to data-driven execution and better data management which mitigate I/O hotspots and minimize resource wastage.

2 Compute-Centric vs. Data-Driven

We begin with an overview of existing data analytics frameworks (§2.1). We then discuss the key design principles of WHIZ (§2.2). Finally, we list the performance issues arising from *compute-centricity* and show how the *data-driven* design adopted by WHIZ overcomes them (§2.3).

2.1 Today: Compute-Centric Engines

Frameworks for batch, graph and streaming analytics rely on execution engines [2, 57]; the engine can be an internal component of the framework or a stand-alone one that the system leverages. The engine is responsible for orchestrating the execution of the analytics job across the cluster.

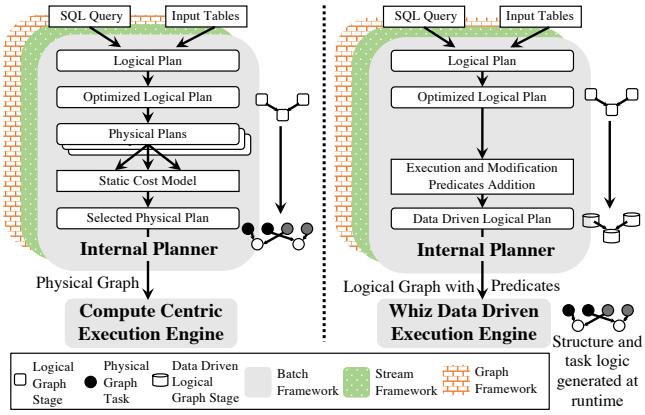


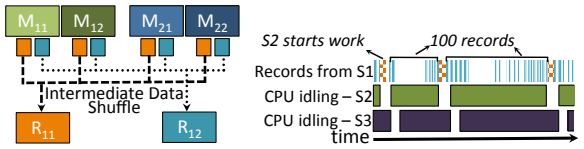
Figure 1: Job Execution Pipelines: Today frameworks hand over physical graphs to the underlying CC execution engine. With WHIZ*, the framework instead hands down a data-driven logical graph and* WHIZ *decides the physical graph at runtime.*

Users submit their jobs to these frameworks (Figure 1) via high-level interfaces (e.g., SQL-like query in case of batch analytics). On submission, the high-level job is handed over to the *internal planner* of the framework which decides the execution plan of the job (expressed in the form of a directed graph). Specifically, the high-level job is translated to a *logical graph* in which different vertices represent different compute *stages* of the overall job and edges represent the dependencies. The logical graph may optionally undergo further optimizations (e.g., to decide the execution order of the stages) and is finally converted to a *physical graph* by undergoing physical optimizations during which low-level execution details such as number of *tasks* per stage (parallelism), dependencies between tasks, resource needs and exact task processing logic are decided.

The execution engine takes the physical graph and orchestrates its execution starting with root stages' tasks processing input data to generate *intermediate data*, which is consumed by downstream stages' tasks.

We explain how the execution engine orchestrates the physical graph and its interplay with intermediate data for different analytics. Figure 2a is an example of a simple *batch analytics* job. Here, two tables need to be filtered based on provided predicates and joined to produce a new table. There are 3 stages: two maps for filtering and one reduce to perform the join. Execution proceeds as follows: (1) Map tasks from both the stages execute first 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 consumer 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 *stream analytics* job (e.g., Figure 2b) has a similar model [11,37,58]; the main difference is that tasks in all stages are always running. A *graph analytics job*, in a framework



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

Figure 2: Simplified examples of existing analytics systems.

that relies on the popular message passing abstraction [42], has a similar but simplified model: the different stages are iterations in a graph algorithm, and thus all stages execute the same processing logic (with the input being the output of the previous iteration).

Compute-centricity: Today's execution engines *early-bind* to a physical graph at job launch-time. Their primary goal is to split up and distribute computation across machines. The composition of this distributed computation, in terms of physical tasks and their dependencies, 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 second class entity as important aspects of job execution such as parallelism, processing and scheduling logic are decided without taking it into account (§2.3).

2.2 WHIZ: A Data-Driven Framework

WHIZ is an *execution engine* that makes intermediate data a first class citizen and supports diverse analytics. WHIZ adopts the following design principles:

1. Decoupling compute and data: WHIZ decouples compute from intermediate data, and the data from all stages across all jobs is written/read to/from a logically separate key-value (KV) datastore (§4), i.e, the datastore resides across the same set of machines on which computations take place. The store is managed by a distinct data management layer called the data service (DS). Similarly, an execution service (ES) manages compute tasks.

2. Programmable data visibility: The above separation enables low-overhead and scalable approaches to gain visibility into *all* runtime data (§5.1). WHIZ DS allows gathering custom runtime properties of intermediate data, via narrow, well-defined APIs.

3. Runtime physical graph generation: During the job execution pipeline, WHIZ skips physical optimization (Figure 1) and thus, *does not early-bind* to a physical graph. Instead, the framework's internal planner performs *data-driven embellishment* on the logical graph to give a *data-driven logical graph*. This embellishment adds predicates to decide when and what logic should be used to process data of each stage, and gives WHIZ the ability to incrementally generate the physical graph at runtime (§6).

4. Data-driven computation: Building on data visibility and data-driven logical graphs, WHIZ initiates data-driven computation by notifying applications when intermediate data predicates within each stage are satisfied (§5.2). Data properties drive all further aspects of computation: task logic, parallelism and sizing (§6).

2.3 Overcoming Compute-centricity Issues

We contrast WHIZ with compute-centricity along flexibility, performance, efficiency, placement, and isolation.

Data opacity, and compute rigidity: In compute-centric frameworks, there is no visibility into intermediate data of a job and the tasks' computational logic are decided a priori. This prevents adapting the tasks' logic based on their input data. Consider the job in Figure 2a. Existing frameworks determine the type of join for the *entire* 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. On the other hand, having fine-grained visibility into input data for each task enables dynamically determining the type of join to use for *different* reduce tasks. A task can use hash join if the total size of *its* input is less than the available memory, and merge join otherwise. WHIZ enables deciding the logic at runtime through its ability to provide visibility and incrementally generate the physical graph (§6.1).

Static Parallelism, Partitioning: Today, jobs' per-stage parallelism, inter-task edges and intermediate data partitioning strategy are decided independent of runtime data and resource dynamics. In Spark [57] the number of tasks in a stage is determined a priori by the user application or by SparkSQL [10]. A hash partitioner is used to place an intermediate (k, v) pair into one of $|tasks|$ buckets. Pregel [42] vertex-partitions the input graph; partitions do not change during the execution of the algorithm.

This limits adaptation to *resource flux* and *data skew*. A running stage cannot utilize newly available compute resources [26, 27, 41] and dynamically increase its parallelism. If some key in a partition has an abnormally large number of records to process, then the corresponding task is significantly slowed down [14], affecting both stage and overall job completion times.

By not early-binding, WHIZ can decide task parallelism and task size based on resources available and data volume. This controls data skew, and provisions task resources proportional to the data to be processed (§6.2).

Idling due to compute-driven scheduling: Modern schedulers [23, 52] decide when to launch tasks for a stage based on the static computation structure. When a stage's computation is commutative+associative, schedulers launch its tasks once 90% of all tasks in upstream stages complete [5]. But the

remaining 10% of producers can take long to complete [14], resulting in tasks idling.

Idling is worse in streaming, where consumer tasks are continuously waiting for data from upstream tasks. E.g., consider the streaming job in Figure 2b. Stage 2 computes and outputs the median for every 100 records received. Between computation, S2's tasks stay idle. As a result, the tasks in the downstream S3 stage *also* stay idle. To avoid idling, tasks should be scheduled *only when, and only as long as, relevant input is available*. In our example, computation should be launched only after ≥ 100 records have been generated by an S1 task. Likewise, in batch analytics, if computation is commutative+associate, it is beneficial to "eagerly" launch tasks to process intermediate data whenever enough data has been generated to process in one batch, and exit soon after it's done.

Idling is easily avoided with WHIZ as it does data-driven scheduling: launches tasks only when predicates are met, i.e, relevant data has been generated (§5.2).

Placement, and storage isolation: Because intermediate data is spread across producer tasks' locations, it is impossible to place *consumer tasks* in a data-local fashion. Such tasks are placed at random [21] and forced to engage in expensive shuffles that consume a significant portion of job runtimes (∼30% [20]).

Also, when tasks from multiple jobs are collocated, it becomes difficult to isolate their hard-to-predict intermediate data I/O. Tasks from jobs generating large intermediate data may occupy much more local storage and I/O bandwidth than those generating less.

Since the WHIZ store manages data from all jobs, it can enforce policies to organize data to meet *per-job* objectives, e.g., data locality for *any* stage (not just input-reading stages), and to meet *cluster* objectives, such as I/O hotspot avoidance and cross-job isolation (§4).

3 WHIZ Overview

We now describe the end-to-end control flow in WHIZ. The end-user submits the job through a high-level interface exposed by the application-specific framework. The framework's internal planner converts the job into a data-driven logical graph through *data-driven embellishment* during which each stage in the graph is annotated with *execution predicates* and *modification predicates*.

Execution predicates determine when data generated by the current stage can be consumed by its downstream stages (e.g., start downstream processing when number of records cross a threshold). Modification predicates determine which processing logic should be chosen at runtime (e.g., decide the join algorithm for the task, say, sort-merge join or hash join) based on data properties.

This data-driven logical graph (e.g., directed acyclic graph in case of batch analytics) , that is expressed via WHIZ APIs (Appendix. A), is submitted to WHIZ via a client (Figure 3).

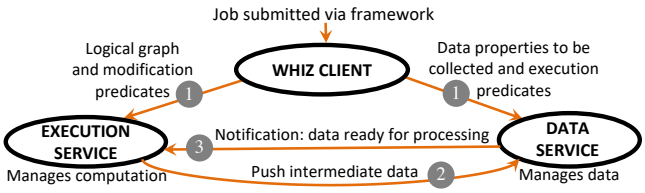


Figure 3: WHIZ *control flow.*

The WHIZ client is the primary interface between the framework running atop WHIZ and the core WHIZ services - the DS and the ES. The client provides the DS with details regarding data properties to be collected and execution predicates. The client also transfers the logical graph and modification predicates to the ES (step 1).

The ES runs the first stage(s) of the logical graph and writes its output to the datastore (step 2). The DS stores the received data and when the execution predicate corresponding to the stage(s) is met, it notifies the ES. The DS piggybacks data statistics (e.g., per-key counts) on this notification to the ES (step 3). On receiving the notification, the ES checks the modification predicates to decide the processing logic and then processes the data. This process repeats. Interactions between the ES, DS and the client are transparent to the framework.

The DS organizes data from all jobs and ensures both per-job and cross-job objectives are met (§4) while simultaneously enabling data visibility through programmable monitoring (§5). The ES, in addition to deciding the processing logic, also determines task parallelism, location and resource use at runtime (§6). In this manner end-users are no longer required to specify low-level details such as task parallelism and data partitioning strategy. In the rest of the paper, we focus on how WHIZ handles data-driven logical graphs and plan to explore designing data-driven embellishers (responsible for embellishment of logical graphs with predicates) that can be added to existing frameworks in the future.

4 Data Store

In WHIZ, all jobs' intermediate data is written to/read from a logically separate datastore (managed by the DS), where it is structured 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. The DS via a cluster-wide master **DS-M** organizes data in the store.

An ideal data organization should achieve three goals: (1) *load balance* and spread all jobs' data, specifically, avoid hotspots and *improve cross-job isolation*, and *minimize within-job skew* in tasks' data processing. (2) maximize job *data locality* by co-locating as much data having the same key as possible. (3) be *fault tolerant* - when a storage node fails, recovery should have minimal impact on job runtime. Our data storage granularity, described next, forms the basis for meeting our goals.

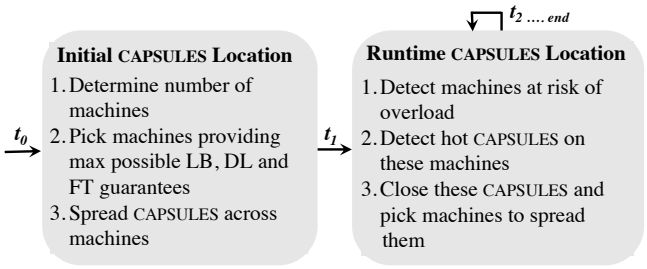


Figure 4: Data organization flow when a stage starts generating data.

4.1 Capsule: A Unit of Data in WHIZ

WHIZ groups intermediate data based on keys into groups called *capsules*. A stage's intermediate data is organized into some large number N capsules; crucially N is *late-bound* as described below, which helps meet our goals above. Intermediate data key range is split N-ways, and each capsule stores all <k, v> data from a given range. WHIZ strives to materialize all capsule data on one machine; rarely (e.g., when there is less space left on a machine), a capsule may be spread across a small number of machines. This materialization property of capsules forms the basis for consumer task data locality.

Furthermore, WHIZ capsule key ranges are *late-bound*: we first determine the set of machines on which capsules from a stage are to be stored; machines are chosen to maximally support isolation, load balance, locality and fault tolerance; the choice of machines then determines the number N for a stage's capsules (§4.2).

Given these machines and N, as the stage produces data at runtime, N capsules are materialized, and dynamically allocated to right-sized tasks; this enables the ES to preserve data-local processing, lower skew, and optimally use compute resources (§6).

4.2 Fast Capsule Allocation

We consider how to place multiple jobs' capsules on machines to avoid hotspots, ensure data locality and minimize job runtime impact on data loss. We formulate an ILP to this end (see Table 7 in Appendix. B). However, solving this ILP at scale can take several tens of seconds delaying capsule placement. WHIZ instead uses a practical approach for the capsule placement problem. First, instead of jointly optimizing global placement decisions for all the capsules, WHIZ solves a "local" problem of placing capsules for each stage independently while still considering inter-stage dependencies; when new stages arrive, or when existing capsules may exceed job quota on a machine, new locations for some of these capsules are determined (see Figure 4). Second, instead of solving a multi-objective optimization, WHIZ uses a linear-time rule-based heuristic to place capsules; the heuristic prioritizes load and locality (in that order) in case machines satisfying all objectives cannot be found. Isolation is always enforced.

Capsule location for new stages (Figure4): When a job j is ready to run, **DS-M** invokes an admin-provided heuristic h1 (Table 1) that assigns job j a quota Q_j per machine. Setting up quotas helps ensure isolation across jobs.

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 //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 load balance (LB), data locality (DL) and maximum possible fault tolerance (FT); 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	*// Given M_v, compute total capsules N.* $N = G$ X M_v, where G = capsules 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 capsules 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.

When a stage v of job j starts to generate intermediate data, **DS-M** 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 $\leq 75\%$ of the quota Q_j for job j. $M_v \geq 2$ ensures opportunities for data parallel processing; a bounded M_v (Table 1) controls the ES task launch overhead (§6.2).

Given M_v, **DS-M** 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 (LB), machines are sorted lightest-load-first, and only ones which have $\leq 75\%$ quota usage for the corresponding job are considered. (2) For data locality (DL), we prefer machines which already materialize other capsules for this stage v, or capsules from other stages whose output will be consumed by same downstream stage as v (e.g., two map stages in Figure 2a). (3) For fault tolerance (FT), we strive to place dependent capsules on different machines to minimize failure recovery time. We pick machines where there are no capsules from any of v's k upstream stages in the job, sorted in descending order of k. 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.

We pick machines from the sub-lists to maximally meet

our objectives in 3 steps: (1) Pick the least loaded machines that are data local and offer as *high fault tolerance as possible* (machines present in all three sub-lists). Note that as we go down the fault tolerance list in search of a total of M_v machines, we trade-off fault tolerance. (2) If despite reaching the minimum possible fault tolerance, i.e., reaching the bottom of the fault tolerance sub-list – the number of machines picked falls below M_v, we completely trade-off fault tolerance and pick the least loaded machines that are data local. (3) If still the number of machines picked falls below M_v, we simply pick the least-loaded machines and trade-off data locality too.

Finally, given $\overrightarrow{M_v}$, **DS-M** invokes h4 and instantiates a fixed number (G) of capsules per machine leading to total capsules per-stage (N) to be $G \times M_v$. While a large G would aid us in better handling of skew and computation as the capsules can be processed in parallel, it comes at the cost of significant scheduling and storage overheads. We empirically study the sensitivity to G (in §9.4); based on this, our prototype uses $G = 24$.

New locations for existing capsules: Data generation patterns can vary across different stages, and jobs, due to heterogeneous compute logics and data skew. Thus a job j may run out of its Q_j on machine m, leaving no room to grow already-materialized capsules of job j on m. **DS-M** reacts to such dynamics by determining, $\forall j$: machines where job j is using $\geq 75\%$ Q_j (h5), closing capsules that are significantly larger or have a higher growth rate than others on such machines (h6), and invokes heuristic h3 to compute the machines to spread these capsules. This focuses on capsules that contribute most load to machines at risk of being overloaded and thus bounds the number of capsules that will spread out.

5 Data Visibility

We now describe how WHIZ offers programmable data monitoring via the DS (§5.1), and how it initiates data-driven computations using execution predicates (§5.2).

5.1 Data Monitoring

Given that intermediate data properties form the basis of data-driven computation, native support for data monitoring is extremely crucial. The DS through its data organization simplifies monitoring as it consolidates a capsule at one or a few locations rather than it being spread across the cluster (§4.1). WHIZ achieves scalable monitoring via per-job masters **DS-JMs** which track light-weight properties related to their capsules.

WHIZ supports *built-in* and *custom* monitors that gather properties per capsule. They are periodically sent to the relevant **DS-JM**. Built-in monitors constantly collect *coarse-grained* properties such as current capsule size, total or number of unique (k,v) pairs, location(s) and rate of growth; apart from being used for data-driven computation, these are used in runtime data organization (§4.2).

Custom monitors are UDFs (user defined functions) that

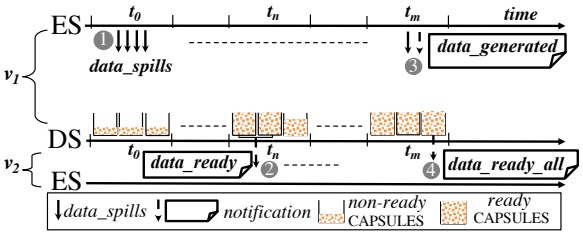


Figure 5: Data-driven computation facilitated by notifications. (1) Intermediate data (v_1) batches sent from ES to DS. (2) DS detects that 2 capsules are ready and sends data_ready notification from DS to ES leading to downstream computation (v_2). (3) ES sends data_generated notification to DS when entire output of v_1 pushed to DS. (4) DS sends data_ready_all notification to ES indicating that all data_ready notifications have been sent.

are used to get *fine-grained* data properties per the job specification. We restrict UDFs to those that can execute in linear time and O(1) state, such as (a) number of entries s.t. values are $<, =$, or $>$ than a threshold; (b) min, max, avg. of keys; and (c) whether data is sorted or not.

Getting visibility into intermediate data through monitors enables data-driven computation as we describe next.

5.2 Indicating Data Readiness

The DS is responsible for initiating data-driven computation. The DS achieves this via two key abstractions: *notifications* and *execution predicates*. The decoupled DS and ES interact via notifications which enable, and track progress of, data-driven computation. Execution predicates enable the **DS-JM** to decide when capsules can be deemed ready for corresponding computation to be run on them.

Notifications: WHIZ introduces 3 types of notifications: (1) A *data_ready* notification is sent by the **DS-JM** to the ES whenever a capsule becomes ready (as per the execution predicate) to trigger corresponding computation. (2) A *data_generated* notification is sent by the ES to the **DS-JM** when a stage finishes generating all its intermediate output. This notification is required because the **DS-JM** is unaware of the number of tasks that the ES launches corresponding to a stage, and thus cannot determine when a stage is completed. (3) A *data_ready_all* notification is sent by the **DS-JM** to the ES when a stage has received all its input data (occurs when *data_ready* notifications regarding all ready input capsules are sent). This notification is required because the ES is unaware of the total number of capsules that the DS deems ready.

The use of these notifications is exemplified in Figure 5. Here: ❶ when a stage v_1 generates a batch of intermediate data, a *data_spill* containing the data is sent to the data store, which accumulates it into capsules (t_0 through t_m). ❷ Whenever the **DS-JM** determines that a collection of v_1's capsules (2 capsules in Figure 5 at t_n) are ready for further processing, it sends a *data_ready* notification per capsule to the ES; the ES launches tasks of a consumer stage v_2 to process such

capsules. This notification carries per-capsule information such as: a list of machine(s) on which each capsule is spread, and a list of statistics collected by the data monitors. ❸ Finally, a *data_generated* notification – from the ES, generated upon v_1 computation completion – notifies the **DS-JM** that v_1 finished generating *data_spills*. ❹ Subsequently, **DS-JM** notifies the ES via the *data_ready_all* event, that all capsules corresponding to v_1 have sent their data_ready events (at t_m). This enables the ES to determine when the immediate downstream stage v_2, that is reading the data generated by v_1, has received all of its input data.

Execution predicates: The interaction between ES and DS via notifications is initiated by execution predicates whose logic is based on the properties collected by the monitors. Each job stage is typically associated with an execution predicate as indicated by the input program, which is transferred to the **DS-JM** by the WHIZ client. If not, default analytics-specific predicates are applied. WHIZ supports diverse execution predicates such as:

1. Data Generated: This predicate deems capsules ready when the computation generating them is done; this is the default predicate for batch and graph analytics in WHIZ; akin to a barrier in batch systems today and bulk synchronous execution in graph analytics.

2. Record Count $\geq X$: The vanilla version of this predicate deems a capsule ready when it has $\geq X$ records from producers tasks; this is the default predicate for streaming systems in WHIZ; akin to micro-batching in existing streaming systems [58], with the crucial difference that the micro-batch is not wall clock time-based, but is based on the more natural intermediate data count.

This predicate can be extended to support pipelining via *ephemeral compute*, i.e, compute is launched once there is partial data and just for the processing duration. The ability to launch compute ephemerally is particularly useful under heavy resource contention. Ephemeral compute can be used to speed up jobs across analytics if they contain commutative+associative operations (§9).

For example, consider the partial execution of a batch (or graph) analytics job, consisting of the first two logical stages (likewise, first two iterations) $v_1 \rightarrow v_2$. If the processing logic in v_2 contains commutative+associative operations, it can start processing its input before all of it is in place. Using this predicate, a capsule generated by v_1 is ready whenever the number of records in it reaches a threshold X. This enables the ES to overlap v_2's computation with v_1's data generation as follows: (1) Upon receiving a *data_ready* notification from the **DS-JM** for capsules which have $\geq X$ records, the ES launches ephemeral tasks of v_2. (2) Tasks read the current data, compute the associative+commutative function on the (k,v) data read, push the result back to data store (in the same capsules advertised through the received *data_ready* notification) and immediately quit. (3) The **DS-JM** waits for each capsule to grow back beyond threshold X for generating subsequent *data_ready* notifications leading to ephemeral tasks being launched again. (4) Finally, when a *data_generated* notification is received from v_1, the **DS-JM** triggers a final *data_ready* notification for all the capsules generated by v_1, and a subsequent *data_ready_all* notification, to enable v_2's final output to be written in capsules and fully consumed by a downstream stage, say v_3 (similar to Figure 5).

This predicate can be further extended to *across* capsules, i.e., the **DS-JM** could deem all capsules ready when the number of entries generated across all capsules cross a threshold. In streaming, such predicates help improve efficiency and performance as ephemeral tasks are launched only when the required input records have streamed into the system and quit post processing (§9.1.3). On the other hand, systems today lack support for ephemeral compute and are forced to deploy long-standing tasks.

3. Special Records: This predicate deems all output capsules of a stage ready on observing a special record in any one capsule. Stream processing systems often rely on "low watermark" records to ensure event-time processing [19, 40], and to support temporal joins [40]. Such predicates can be used to launch, *on demand*, temporal operators whenever a low watermark record is observed at any of a stage's output capsules. In contrast, systems today have the operators always running and this leads to compute idling when there are no records to process.

6 Execution Service

While the DS initiates data-driven computation by notifying when data is ready for processing, the ES carries out all other data-driven execution aspects by *incrementally generating the physical graph*. It does so via a per-job master **ES-JM** that given ready capsules, and available resources[1]: (a) determines the appropriate processing logic to use (§6.1); (b) determines optimal parallelism and deploys tasks to minimize skew and shuffle; (c) maps capsules to tasks in a resource-aware fashion (§6.2).

6.1 Selecting Compute Logic

Upon detecting a ready capsule, the **DS-JM** sends the *data_ready* notification, with capsule properties (including fine-grained ones) piggybacked, to the **ES-JM**. The **ES-JM** then uses modification predicates associated with this stage to determine the exact processing logic.

Modification predicates give the ability to decide processing logic at runtime based on the received data properties and available resources. Importantly, WHIZ also provides jobs with the flexibility to use *different* processing logic for different input capsules of the same stage. For e.g., consider a batch analytics job that involves joining two tables.[2] In such

[1] Similar to existing frameworks, a cluster-wide Resource Manager decides available resources as per cross-job fairness.

[2] DS via per-job quotas ensures that the input tables use the same # of capsules and because both tables use the same key, i.e., the join key, while

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

Table 2: Heuristics to group capsules and assign them to tasks.

a scenario, the SQL framework running atop sets the modification predicates of the join stage to choose the appropriate join algorithm between, say, sort-merge join[3] and hash join[4] as follows: (a) if both capsules are already sorted, and the max value in the first capsule is less than the min value in the other capsule (no intersection), then skip unnecessarily launching a task to do the join; (b) if the size of one capsule is significantly smaller than the other one (and data is not sorted), then use hash join (as it is typically less expensive to create a hash table of the smaller capsule, than sorting the large one); and (c) if data is sorted or (a)–(b) don't satisfy, then default to sort-merge join.

Crucially, such predicates enable stream jobs submitted to WHIZ, to change their processing logic *over time*, as opposed to being early bound to processing logic (status-quo today). For e.g., predicates allow a job involving temporal join to change its join algorithm over time and choose the appropriate one, from the three choices (a)–(c) above, based on data properties and available resources.

6.2 Task Parallelism, Placement, and Sizing

Given a set of ready capsules (C) for a stage, the ES-JM needs to map capsules to tasks, and determine their location (across machines M) and sizes (resources) so as to minimize cross-task skew and shuffle while taking available resources into account. To do so, we propose an *iterative procedure* that applies a set of heuristics (Table 2) repeatedly until tasks for all ready capsules are allocated, and their locations and sizes determined.

The iterative procedure consists of 3 steps: (a) generate optimal subsets of capsules to minimize cross-subset skew (using h7), (b) decide on which machine should a subset be processed to minimize shuffle (h8), and (c) determine resources required to process each subset (h9).

First, we group capsules C into a collection of *subsets* $\overrightarrow{C}$ using h7. We then try to assign each group to a task. Our grouping into subsets attempts to ensure that data in a subset is spread on just one or a few machines (lines (b.i-b.iii)), which minimizes shuffle, and that the data is spread roughly evenly across subsets (line (b.iv)) making cross-task performance uniform. We place a bound $CaMax$, equaling twice the size of the largest capsule, on the total size of a subset (see line (a)). This ensures that multiple (at least 2) capsules are present in each subset and allows mitigating stragglers by *only* assigning the yet-to-be-processed capsules to the speculative task.[5]

Second, we determine a preferred machine to process each subset using h8; this is a machine where most if not all capsules in the subset are materialized (line (d)). Choosing a machine in this manner minimizes shuffle.

Finally, given available resources across the preferred machines (from the cluster-wide resource manager [52]) we need to allocate tasks to process subsets. But some machines may not have resource availability. For the rest of this iteration, we ignore such machines and the subsets of capsules that prefer such machines.

Given machines with resources $\overrightarrow{A}$, we assign a task for each subset of capsules which can be processed, and allocate task resources *altruistically* using h9. That is, we first compute the minimum resource available to process unit data (F; line (f)). Then, for each task, the resource allocated (line (g)) is F times the total data in the subset of capsules allocated to the task ($\lvert \overrightarrow{C}[i] \rvert$).

Allocating task resources proportional to input size ensures that tasks have similar finish times. Allocating resources corresponding to the minimum available helps further: if a job gets more resources than what is available for the most constrained subset, then it does not help the job's completion time (which is determined by the most constrained subset's processing). Altruistically "giving back" such resources speeds up other jobs/stages.

The above 3 steps repeat whenever new capsules are ready, or existing ones can't be scheduled. Similar to delay scheduling [56], we attempt several tries to execute a group which couldn't be scheduled on its preferred machine due to resource unavailability, before marking capsules *conflicting*. These are

writing to the store, key-range split for both tables is the same.

[3] This (a) sorts the two input capsules on the join key and (b) merges them by comparing the records.

[4] This (a) builds a hash table on the join key using the smaller capsule and (b) probes for matches using the other capsule.

[5] Speculative tasks today [12–14, 38, 59] reprocess the entire input leading to duplicate work and resource wastage.

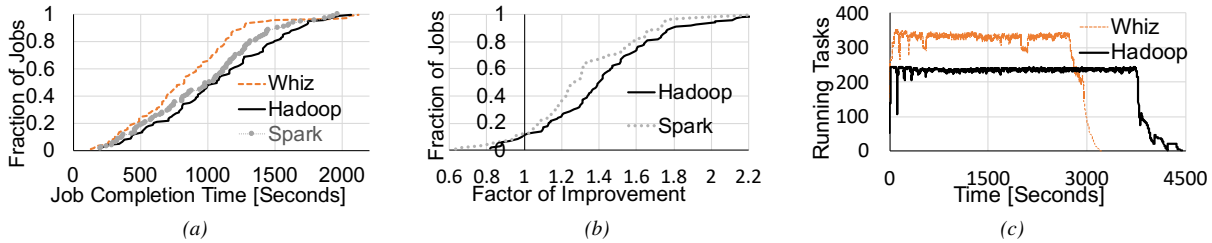


Figure 6: [Batch Analytics] (a) CDF of JCT; (b) CDF of factors of improvement of individual jobs using WHIZ *w.r.t. baselines; (c) Snapshot of running tasks during one of the experiments. Gains are lower w.r.t. Spark due to our Hadoop-based implementation, and thus using a non-optimized in-memory store.*

re-grouped in the next iteration (line (b.v)). Finally, capsules that cannot be executed under any grouping are marked troublesome (line (b.vi)) and processed on any machine (line (d.ii)).

7 Fault Tolerance

Task Failure. When a task fails due to a machine failure, only the failed tasks need to be re-executed if the input capsules are not lost. But, this will result in duplicate data in all capsules for the stage leading to data inconsistencies. To address this, we use checksums at the consumer task-side WHIZ library to suppress duplicate data.

However, if the failed machine also contains the input capsules of the failed task, then the **ES-JM** triggers the execution of the upstream stage(s) to regenerate the input capsules of the failed task. Recall that WHIZ's fault tolerance-aware capsule storage (§4) helps control the number of upstream (ancestor) stages that need to be re-executed in case of data loss.

DS-M/DS-JM/ES-JM. WHIZ maintains replicas of DS-M/DS-JM daemons using Apache Zookeeper [28], and fails over to a standby. Given that WHIZ generates the physical graph of a job at runtime in a data-driven manner, upon ES-JM failure, we simply need to restart it so that it can resume handling notifications from the DS. During this time already launched tasks continue to run.

8 Implementation

We prototyped WHIZ by modifying Tez [5] and leveraging YARN [52]. The DS, implemented from scratch, has three kinds of daemons (managed via YARN): cluster-wide master **DS-M**, per-job masters **DS-JM** and workers **DS-W**. DS-M does data organization across DS-Ws. DS-JMs collect statistics and notify ES-JM when execution predicates are met. DS-Ws run on cluster machines and do node-level management: (a) store data received from ES/other DS-Ws in local in-memory file system (`tmpfs` [48]) and transfer data to other DS-Ws per DS-M directives; (b) report statistics to DS-M/DS-JMs via heartbeats; and (c) provide ACK to tasks for data written.

The ES was implemented by modifying Tez. It consists of per-job masters **ES-JM** which are responsible for generating the physical graph at runtime. 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 WHIZ client is a standalone process per-job.

All communication (asynchronous) between DS, ES and client is through RPCs in YARN using Protobuf [8]. We also use RPCs between the YARN Resource Manager (RM) and ES-JM to propagate resource allocations (§6).

9 Evaluation

We evaluated WHIZ on a 50-machine cluster deployed on CloudLab [6] using publicly available benchmarks – *batch* TPC-DS jobs, PageRank for *graph analytics*, and synthetic *streaming* jobs. Unless otherwise specified, we set WHIZ to use default execution predicates, equal storage quota ($Q_j = 2.5GB$) and 24 capsules per machine.

9.1 Experiment Setup

Workloads: We consider a mix of jobs, all from TPC-DS (**batch**), or all from PageRank (**graph**). For **streaming**, we use a variety of different queries described in detail later. 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. We run each experiment thrice and present the median.

Cluster, baseline, metrics: Machines have 8 cores, 64GB memory, 256GB storage, and a 10Gbps NIC. We compare WHIZ as follows: (1) **Batch**: *vs.* Tez [5] running atop YARN [52], for which we use the shorthand "Hadoop" or "CC"; and *vs.* SparkSQL [15]; (2) **Graph**: *vs.* Giraph (i.e., open source Pregel [42]); and *vs.* GraphX [24]; (3) **Streaming**: *vs.* SparkStreaming [58].

For a fair comparison, we ensure Hadoop/Giraph use `tmpfs`. We study the relative improvement in the average job completion time (JCT), or $JCT_{CC}/JCT_{\text{WHIZ}}$. We measure efficiency using *makespan*.

9.1.1 Batch Analytics

Performance and efficiency: Figure 6a shows the JCT distributions of WHIZ, Hadoop, and Spark for the TPC-DS work-

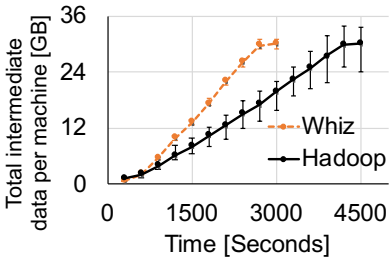


Figure 7: Cross-job average, min *and* max *intermediate data per machine during one of our batch analytics experiments.*

load. Only 0.4 (1.2) highest percentile jobs are worse off by $\leq 1.06\times$ ($\leq 1.03\times$) than Hadoop (Spark). WHIZ speeds up jobs by **1.4×** (**1.27×**) on average, and **2.02×** (**1.75×**) at 95*th* percentile w.r.t. Hadoop (Spark). Also, WHIZ improves makespan by **1.32×** (**1.2×**).

Figure 6b presents improvement for individual jobs. For more than 88% jobs, WHIZ outperforms Hadoop and Spark. **Only 12**% jobs slow down to $\leq 0.81\times$ ($0.63\times$) using WHIZ. Gains are $> 1.5\times$ for > 35% jobs.

Sources of improvements: We observe that *more rapid processing* due to *data-driven execution*, and *better data management* contribute most to benefits.

First, we snapshot the number of running tasks across all the jobs in one of our experiments when running WHIZ and Hadoop (Figure 6c). WHIZ has $1.45\times$ more tasks scheduled over time which translates to jobs finishing $1.37\times$ faster. It has **1.38×** better cluster efficiency than Hadoop. Similar observations hold for Spark (omitted).

The main reasons for rapid processing/high efficiency are: (1) The DS 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) Based on DS-provided properties, ES's data-driven actions provide similar input sizes for tasks in a stage – within 14.4% of the mean.

Second, Figure 7 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. WHIZ mitigates hotspots (§4) improving overall performance.

We observe jobs generating less intermediate data are more prone to performance losses in WHIZ, especially under ample resource availability as WHIZ strives for capsule-local task execution (§6.2). If resources are unavailable, WHIZ will assign the task to a data-remote node, or get penalized waiting for data-local placement.

9.1.2 Graph Processing

We run multiple PageRank (40 iterations) jobs on the Twitter Graph [17, 18]. In each iteration, vertices run the processing logic and exchange their output as messages with each other. WHIZ groups messages into capsules based on vertex ID. We

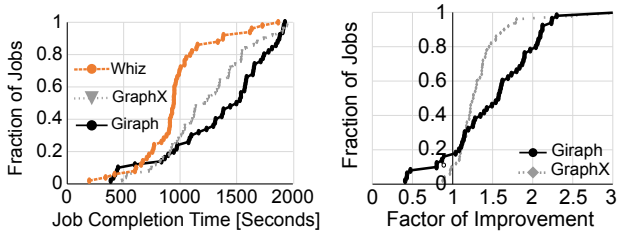


Figure 8: [Graph Analytics] (a) CDF of JCT using WHIZ*, GraphX and Giraph; (b) CDF of factors of improvement of individual jobs using* WHIZ *w.r.t. GraphX and Giraph.*

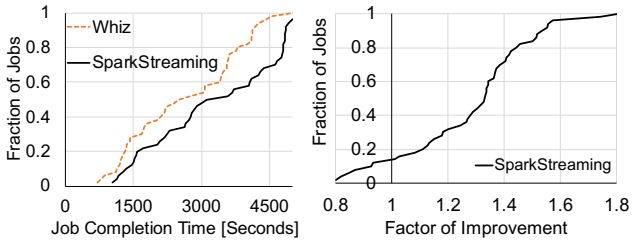


Figure 9: [Stream Analytics] (a) CDF of JCT using WHIZ *and SparkStreaming; (b) CDF of factors of improvement of individual jobs using* WHIZ *w.r.t. SparkStreaming.*

use an *execution predicate* that deems a capsule ready when ≥ 1000 messages are present.

Figure 8a shows the JCT distribution of WHIZ, GraphX and Giraph. WHIZ speeds up jobs by **1.33×** (**1.57×**) on average and **1.57×** (**2.24×**) at the 95*th* percentile w.r.t. GraphX (Giraph) (Figure 8b). Gains are lower w.r.t. GraphX, due to its efficient implementation atop Spark. However, **< 10**% jobs are slowed down by $\leq 1.13\times$.

Improvements arise for two reasons. First, WHIZ is able to *deploy appropriate number of ephemeral tasks*: execution predicates immediately indicate data availability, and runtime parallelism (§6.2) allows messages to high-degree vertices [24] to be processed by more than one task. Also, WHIZ has $1.53\times$ more tasks (each runs multiple vertex programs) scheduled over time; rapid processing and runtime adaptation to data directly leads to jobs finishing faster. Second, because of ephemeral compute, WHIZ doesn't hold resources for a task if not needed, resulting in **1.25×** better cluster efficiency.

9.1.3 Stream Processing

We run multiple stream jobs, each calculating top 5 common words for every 100 distinct words from synthetic streams replaying GBs of text data from HDFS.

Spark Streaming discretizes the records stream into time-based micro-batches and processes every micro-batch duration. We configure the micro-batch interval to 1 minute. With WHIZ, given the semantics of the processing logic, we use an

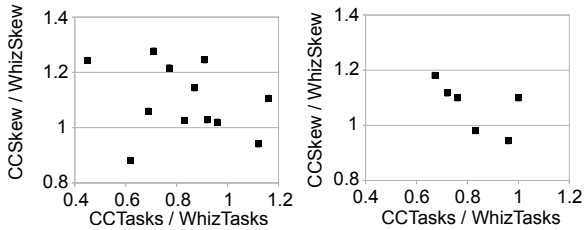


Figure 10: Hadoop w.r.t. WHIZ *fraction of tasks allocated vs. the fraction of skew in a given stage: (a) for a job with* 12 *stages where* WHIZ *improves JCT by* 1.6×*; (b) for a job with* 6 *stages, where* WHIZ *improves JCT by* 1.2×. $\frac{CCSkew}{WhizSkew} > 1$ *means* WHIZ *has less skew;* $\frac{CCTasks}{WhizTasks} < 1$ *means Hadoop under-parallelizes.*

execution predicate to enable computation whenever ≥ 100 distinct records are present.

Figures 9a, 9b show our results. WHIZ speeds up jobs by **1.33×** on average and **1.55×** at the 95*th* %-ile. Also, 15% of the jobs are slowed down to around **0.8×**.

The gains are due to *data-driven computation* via execution predicates; WHIZ does not have to delay execution till the next micro-batch if data can be processed now. A Spark Streaming task has to wait as it has no data visibility. In our experiments, more than **73**% executions happen at less than 40*s* time intervals with WHIZ.

Additionally, we evaluate the role of modification predicates in streaming in §9.2.

9.1.4 WHIZ Overheads

CPU, memory overhead: We find that DS-W (§8) processes inflate the memory and CPU usage by a negligible amount even when managing data close to storage capacity. DS-M and DS-JM have similar resource profiles.

Latency: We compute the average time to process heartbeats from various ES/DS daemons, and WHIZ client. For 5000 heartbeats, the time to process each is $2-5ms$. We implemented the WHIZ client and ES-JM logic atop Tez AM. Our changes inflate AM decision logic by $\leq 14ms$ per request with negligible increase in AM memory/CPU.

Network overhead from events/heartbeats is negligible.

9.2 Benefits of Data-driven Computation

The overall benefits above included the effects of execution predicates and incrementally generating the physical graph. We now delve deeper to further shed light into late-binding benefits.

Skew and parallelism: Figure 10 shows fractions of skew and parallelism as generated by Hadoop w.r.t. WHIZ for two TPC-DS jobs from one of our runs. WHIZ's ability to dynamically change parallelism at runtime, driven by the number of capsules for each vertex, leads to significantly less data skew than Hadoop. When Hadoop is under-parallelizing, the skew is significantly higher than WHIZ (up to 1.43×). Over-parallelizing does not help either; Hadoop incurs up to 1.15× larger skew, due to its rigid data partitioning and tasks allocation schemes. Even when WHIZ incurs more skew (up to 1.26×), corresponding tasks will get allocated more resources to alleviate this overhead (§6.2).

% Skew	Improvement Factor
10%	1.1
30%	1.47
50%	1.87
70%	2.48
90%	2.67

Table 3: Improvement in cumulative time using modification predicates w.r.t no predicates. Predicate chooses hash join if the ratio of input capsules' sizes is ≥ 3.

Modification predicates: To evaluate the benefits enabled by WHIZ's ability to late-bind processing logic, we pick a query from our TPC-DS workload which has a join and run it with and without modification predicates while varying the skew between the input tables. Modification predicates allow the job to pick the join algorithm between sort-merge join and hash join (see §6.1) for the different tasks. Table 3 shows the relative improvement in cumulative time (summation over duration of all tasks of the job) with and without predicates (sticks to sort-merge join). We see that predicates improve the cumulative time **1.1×–2.8×** as the skew in capsule sizes increases. This is because with modification predicates, WHIZ chooses to use the hash join when skew between the task inputs exists as building a hashmap on the smaller input is typically cheaper than sorting the other input (occurs when sort-merge join is used instead). Gains increase with skew as the join performance difference also increases.

We also quantify the benefits of modification predicates for stream processing. We run a stream query that performs event-time temporal join over 3 minute intervals (execution predicate indicates to wait for watermark) with and without the above modification predicates. We change skew-% randomly (from 10%, 20%,..., 90%) between the two input sources over the same time interval and observe that using modification predicates leads to 1.7× average improvement in cumulative time.

Additionally, we run microbenchmarks to delve further into WHIZ's data-driven benefits (results in Appendix C).

9.3 Load Balancing, Locality, Fault Tolerance

To evaluate DS load balancing (LB), data locality (DL) and fault tolerance (FT), we stressed the data organization under different cluster load. We used job arrivals and all stages' capsule sizes from one of our TPC-DS runs.

Figure 11 shows that: (1) WHIZ prioritizes load balancing and data locality over fault tolerance across cluster loads (§4.2); (2) when the available resources are scarce (5× higher load than initial), all three metrics suffer. However, the maxi-

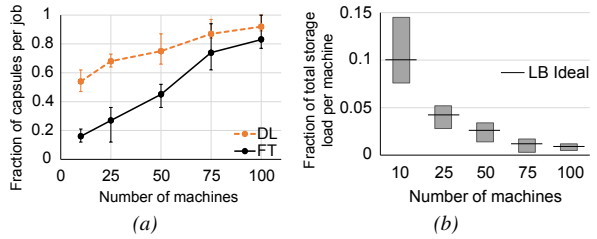


Figure 11: (a) Average, min and max fraction of capsules which are data local (DL) respectively fault tolerant (FT) across all the jobs for different cluster load; (b) Max, min and ideal storage load balance (LB) on every machine for different cluster load.

% Machines Failed	JCT [Seconds]		
	Avg	**Min**	**Max**
None	725	215	2100
10%	740	250	2320
25%	820	310	2360
50%	1025	350	2710
75%	1600	410	3300

Table 4: JCT under random machine failures.

mum load imbalance per machine is $< 1.5\times$ than the ideal, while for any job, $\geq 47\%$ of the capsules are data local. Also, on average 16% of the capsules per job are fault tolerant; (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 capsules are data local, 71% are fault tolerant, with at most $1.17\times$ load imbalance per machine than the ideal.

Failures: Using the same workload, we also evaluated the performance impact in the presence of machine failures (Table 4). We observe that WHIZ does not degrade job performance by more than $1.13\times$ even when 25% of the machines fail. This is mainly due to DS's ability to organize capsules to be fault tolerant across ancestor stages and avoid data recomputations. Even when 75% of the machines fail, the maximum JCT does not degrade by more than $1.57\times$, mainly due to capsules belonging to some ancestor stages still being available, which leads to fast recomputation for corresponding downstream vertices.

9.4 Sensitivity Analysis

Impact of Contention: 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, WHIZ improves over Hadoop by $1.39\times$ ($1.32\times$) on average in terms of JCT (makespan). Even *at high contention* (up to $4\times$), WHIZ's gains keep increasing $1.83\times$ ($1.42\times$). This is because of data-driven execution and better data management which minimizes resource wastage, time spent in shuffling, and leads to few hotspots.

Multiple of Original Load	# Capsules							
	8	**16**	**20**	**24**	**28**	**32**	**36**	**40**
1	1.07	1.33	1.46	1.52	1.57	1.63	1.54	1.46
2	1.10	1.16	1.53	1.58	1.56	1.61	1.47	1.31
4	0.85	1.12	1.34	1.39	1.32	1.16	0.95	0.74

Table 5: Factors of improvement w.r.t. Hadoop for different number of capsules per machine and cluster load.

Impact of G (number of capsules per machine): We now provide the rationale for picking $G = 24$. Table 5 shows the factors of improvements w.r.t. Hadoop for different values of G and levels of contention.

The main takeaways are as follows: for $G = 8$ the performance gap between WHIZ and Hadoop is low ($< 1.1\times$). This is expected because small number of capsules results in less data locality (each capsule is more likely to be spread). Further, the gap decreases at high resource contention. In fact, at $4\times$ the cluster load, Hadoop performs better ($0.85\times$). At larger values of G the performance gap increases. For example, at $G = 24$, WHIZ gains are the most (between $1.39\times$ and $1.58\times$). This is because larger G implies (1) more flexibility for WHIZ to balance the load across machines; (2) more likely that few capsules are spread out; (3) lesser data skew and more predictable per task performance. However, a very large G does not necessarily improve performance, as it can lead to massive task parallelism. The resulting scheduling overhead degrades performance, especially at high load.

Altruism: Assigning resources altruistically is beneficial as it improves median ($95th$ %-ile) JCT by $1.48\times$ ($4.8\times$) for our TPC-DS runs w.r.t a greedy approach where tasks use all of their available resources. Only 16% jobs are slowed down by $\leq 0.6\times$.

10 Related Work

We now discuss the various related efforts to overcome the various limitations of compute-centricity. WHIZ, with its clean separation of compute and intermediate data, overcomes the various limitations of compute-centricity in a unified manner while prior related efforts propose point-fixes to a subset of the limitations that plague compute-centric execution engines.

Data opacity: Almost all database and bigdata SQL systems [10, 12, 54] use statistics computed ahead of time to optimize execution. Adaptive query optimizers (QOs) [22] use dynamically collected statistics and re-invoke the QO to re-plan queries top-down. In contrast, WHIZ alters the query plans on-the-fly at the *execution layer* based on *run-time* data properties, thereby circumventing additional expensive calls to the QO. Tukwila [30] reformulates queries by using run-time visibility in a limited fashion to fix poor statistics maintenance in QOs. WHIZ instead enables much richer visibility and supports a richer set of actions that enable true data-centric behavior. RoPE [12] leverages historical statistics from prior plan executions in order to tune future executions. WHIZ,

instead uses runtime properties.

CIEL [45] is an execution engine that provides support for data-dependent iterative or recursive algorithms by dynamically deciding the execution graph as tasks execute. However, low-level execution aspects such as per-stage parallelism are decided beforehand. Optimus [33] extends frameworks such as CIEL [45] and Dryad [29] to enable runtime logic rewriting and parallelism selection by using streaming-based algorithms to collect aggregated statistics on intermediate data. RIOS [39] is an optimizer for Spark that solely focuses on optimizing joins by deciding the join order and stage-level join implementation using the approximate statistics collected at runtime. While Optimus and RIOS attempt to provide data visibility, neither of them does a clean separation of compute and data; this limits data-local processing and imposes I/O interference as intermediate data organization is determined by the compute structure. Moreover, both still resort to compute-driven scheduling and do not use data visibility to decide if/when tasks should be scheduled. Further, RIOS adopts static per-stage parallelism, and cannot make fine-grained logic changes (e.g., task-level) as table-level statistics are aggregated by a separate Spark job and then sent to the Spark driver which is responsible for making runtime changes. Overall, WHIZ is a general approach to data-driven computation that subsumes all prior efforts, and enables new data-driven execution benefits; its clean separation of data enables data-locality and I/O isolation management.

Skew and parallelism: Some parallel databases [25, 34, 55] and big data systems [38] dynamically adapt to data skew for single large joins. In contrast, WHIZ holistically solves data skew for all joins across multiple jobs. [25, 38] deal with skew in MapReduce by dynamically splitting data for slow tasks into smaller partitions and processing them in parallel. But, they can cause additional data movement from already slow machines leading to poor performance. Hurricane [16] mitigates skew via an adaptive task partitioning scheme by cloning slow tasks at runtime and performing data organization such that all tasks, be it the primary task or its clones, can access data that requires processing. However, Hurricane can lead to additional processing overheads as it does not take data locality into account while organizing data (data corresponding to the same key can be spread across multiple machines) and also involves an additional merge step that combines the partial outputs of the clones using the end-user provided merging logic. Moreover, Hurricane does not have fine-grained visibility into intermediate data and thus cannot do fine-grained task logic changes and still adopts compute-driven scheduling. Henge [32] supports multi-tenant streaming by deciding parallelism based on SLOs. However, it still adopts compute-driven scheduling.

Decoupling: Naiad [44] and StreamScope [53] also decouple intermediate data. They tag intermediate data with vector clocks which are used to trigger compute in the correct order. Both support ordering driven computation, orthogonal to data-driven computation in WHIZ. Also, StreamScope is not applicable to batch/graph analytics. Crail [49] decouples intermediate data from compute so that various execution engines can easily leverage modern storage hardware (including tiered storage) to perform intermediate data management. However, the compute structure still decides the number of partitions across which data is organized. Additionally, it adopts a similar data storage abstraction as well as data placement policy to Hurricane and thus incurs additional overheads as it does not take data locality into account. Moreover, Crail recommends replicating data in case fault tolerance is required which can further lead to additional overheads. Instead, WHIZ provides fault tolerance by intelligent placement of intermediate data so as to minimize recovery time. Also, similar to Hurricane, it does not have fine-grained data visibility to drive all aspects of execution.

Storage inefficiencies: For batch analytics, [31,46] addresses storage inefficiencies by pushing intermediate data to the appropriate external data services (like Amazon S3 [1], Redis [9]) while remaining cost efficient and running on serverless platforms. Similarly, [35] is an elastic data store used to store intermediate data of serverless applications. However, since this data is still opaque, and compute and storage are managed in isolation, these systems cannot support data-driven computation or achieve data locality and load balancing simultaneously.

11 Summary

The compute-centric nature of existing data analytics frameworks hurts flexibility, performance, efficiency, and job isolation. With WHIZ, analytics undergo data-driven execution aided by a clean separation of compute from intermediate data. WHIZ enables monitoring of data properties and using these properties to decide all aspects of execution - what to launch, where to launch, and how many tasks to launch, while ensuring isolation. Our evaluation using batch, stream and graph workloads shows that WHIZ significantly outperforms state-of-the-art.

Acknowledgments

We would like to thank our shepherd, Chris Rossbach, the anonymous reviewers of NSDI'21 and the members of WISR Lab for their insightful comments and suggestions. This research was supported by NSF Grants CNS-1565277, CNS-1719336, CNS-1763810, CNS-1838733 and by gifts from Google and VMware.

References

[1] Amazon S3. https://aws.amazon.com/s3/.

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

[7] Presto | Distributed SQL Query Engine for Big Data. prestodb.io.

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

[9] Redis. https://redis.io/.

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

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

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

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

[14] 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.

[15] 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.

[16] 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.

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

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

[19] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Computer Society TCDE Bulletin*, 36(4), 2015.

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

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

[22] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Found. Trends databases*, 1(1):1–140, Jan. 2007.

[23] 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.

[24] 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.

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

[26] B. Huang, N. W. Jarrett, S. Babu, S. Mukherjee, and J. Yang. Cumulon: Matrix-based data analytics in the cloud with spot instances. *PVLDB*, 9(3):156–167, 2015.

[27] B. Huang and J. Yang. Cumulon-d: Data analytics in a dynamic spot market. *PVLDB*, 10(8):865–876, 2017.

[28] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *ATC*, 2010.

[29] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[30] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An adaptive query execution system for data integration. *ACM SIGMOD Record*, 1999.

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

[32] F. Kalim, L. Xu, S. Bathey, R. Meherwal, and I. Gupta. Henge: Intent-driven multi-tenant stream processing. In *SoCC*, 2018.

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

[34] 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.

[35] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *OSDI*, 2018.

[36] 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.

[37] 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.

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

[39] Y. Li, M. Li, L. Ding, and M. Interlandi. Rios: Runtime integrated optimizer for spark. In *SoCC*, 2018.

[40] W. Lin, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. Streamscope: continuous reliable distributed processing of big data streams. In *NSDI*, 2016.

[41] K. Mahajan, M. Chowdhury, A. Akella, and S. Chawla. Dynamic query re-planning using QOOP. In *OSDI*, 2018.

[42] 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.

[43] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. *JMLR*, 17(1):1235–1241, 2016.

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

[45] 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.

[46] Q. Pu, S. Venkataraman, and I. Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *NSDI*, 2019.

[47] 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.

[48] P. Snyder. tmpfs: A virtual memory file system. In *EUUG Conference*, 1990.

[49] P. Stuedi, A. Trivedi, J. Pfefferle, A. Klimovic, A. Schuepbach, and B. Metzler. Unification of temporary storage in the nodekernel architecture. In *ATC*, 2019.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD*, 2013.

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

[56] 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.

[57] 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.

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

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

A WHIZ APIs

Table 6 shows the various APIs exposed by WHIZ that are used by frameworks running atop WHIZ to submit programs. Similar to APIs today, (1)–(3) APIs are job composition APIs to create logical directed graphs (WHIZ has similar APIs to load a graph). Crucially, WHIZ *does not require* the frameworks to specify low-level details like parallelism and partitioning strategy. The *addCustomMonitor* API is used by the framework to submit UDFs to collect custom statistics (apart from the built-in monitors that WHIZ supports).

During the data embellishment phase, the framework annotates the logical graph with execution and modification predicates using *addExecutionPredicate* API and *addModificationPredicate* API respectively. The framework provides the predicates based on data properties via a UDF.

We now show how to write execution and modification predicates for a number of applications used in our testbed experiments (§9).

Figure 12a shows the execution predicate used to run the PageRank algorithm. Specifically, the UDF specifies that as soon as 1000 messages corresponding to a vertex are received, the data is ready to be processed. Similarly, Figure 12b shows the execution predicate used to run the streaming job that returns the top 5 words when we see 100 distinct words. This predicate indicates that as soon as 100 unique key records globally have been received, the data is ready to be processed.

Lastly, Figure 12c shows how to specify modification predicates to decide the join algorithm on the fly for a batch SQL query as well as a streaming query involving a temporal join. This predicate takes as input the properties collected of the two capsules (from the two input tables) and chooses the join algorithm based on the amount of skew. If the amount of skew is less than the threshold specified by the framework, then sort-merge join is used; otherwise hash join is chosen.

B Allocating Capsules to Machines ILP

We consider how to place multiple jobs' capsules to avoid hotspots, reduce per-capsule spread (for data locality) and minimize job runtime impact on data loss. We formulate a binary integer linear program (see Table 7) to this end. The indicator decision variables, x_i^k, denote that all future data to capsule g_k is materialized at machine M_i. The ILP 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 capsules. Minimizing this ensures load balance and avoids hotspots. The second part (O_2) represents the sum of *data-spread penalty* across all capsules. Here, for each capsule, we define the primary location as the machine with the largest volume of data for that capsule. 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 capsules. 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 capsule 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.

However, solving this ILP at scale can take several tens of seconds delaying capsule placement. Thus, WHIZ uses a linear-time rule-based heuristic to place capsules (as described in §4).

C WHIZ Microbenchmarks

Apart from the experiments on the 50-machine cluster (§9), we also ran several microbenchmarks to delve deeper into WHIZ's data-driven benefits. The microbenchmarks were run on a 5 machine cluster and the workloads consists of the following jobs: $\mathbb{J}_1$ ($v_1 \rightarrow v_2$) and $\mathbb{J}_2$ ($v_1 \rightarrow v_2 \rightarrow v_3$). These patterns typically occur in TPC- DS queries.

Skew and parallelism: Figure 13a shows the execution of one of the $\mathbb{J}_2$ queries from our workload when running WHIZ and CC. WHIZ improves JCT by 2.67× 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, WHIZ ends up generating capsules that are approximately equal in size and decides at runtime a max. input size per task of $1GB$ (twice the largest capsule). This leads to running 5 tasks of v_2 with equal input size and 2.1× 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 Figure 13a) and completion time for v_3 suffers (85s). In contrast, WHIZ assigns at runtime only 5 tasks of v_3 which can run in a single wave; v_3 finishes 1.23× 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, which was assigned 2 capsules by WHIZ.

In CC (Figure 13b), 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 WHIZ, upon straggler detection, the straggler (v_2) is notified to finish processing the current capsule; a task v_2' is launched and assigned data

	API	Description
1	createJob(name:Str, type:Type)	Creates a new job which can be of type BATCH, STREAM or GRAPH.
2	createStage(j:Job, name:Str, impl:StageImpl, prop:StageProperties)	Adds a logical stage of to a Job with a default processing logic. StageProperties specifies properties of the logic (e.g., if it is commutative+associative).
3	addDependency(j: Job, s1: Stage, s2: Stage)	Adds a starts before relationship between stages s1 and s2.
4	addCustomMonitor(j:Job, s:Stage, impl:DataMonitorImpl)	Adds a custom data monitor to compute statistics over data generated by stage s. DataMonitorImpl is a UDF.
5	addExecutionPredicate(j:Job, s:Stage, predicates:ExecutionPred)	Decides when downstream stages can consume current stage's data based on the predicates specified by ExecutionPred. ExecutionPred is used by DS to decide when data is ready for processing.
6	addModificationPredicate(j:Job, s:Stage, predicates:ModifyPred)	Decides processing logic for the input capsules that are ready based on the predicates specified by ModifyPred. ModifyPred is used by the ES to decide which processing logic to use based on the data properties from the DS.

Table 6: WHIZ APIs - Used by the frameworks running atop WHIZ to translate the high-level job submitted by end users to WHIZ-compatible data-driven logical graphs.

```
1 def ExecutionPredicate():
2   for key in keys:
3     if (DS.monitor.num_entries(key) >= 1000):
4       return true
5   return false
```

```
1 def ExecutionPredicate():
2   if (DS.monitor.global_unique_entries >= 100):
3       return true
4   return false
```

```
1 def ModificationPredicates(capA, capB):
2   // THRESHOLD is set by the framework
3   sizeRatio = max(capA.size, capB.size)/ min(capA.size, capB.size)
4   if (sizeRatio >= THRESHOLD):
5     return HashJoinImpl //Refers to hash join
6   return SortMergeJoinImpl //Refers to sort-merge join
```

Figure 12: Examples of predicates. (a) Execution predicate for the PageRank algorithm - deem capsule ready when it has 1000 messages, (b) Execution predicate for the streaming application - deem capsules ready when we see 100 unique entries and (c) Modification predicate for changing join algorithm on the fly.

Objectives (to be minimized):

O_1	$\max_i \left(\sum_k (b_i^k + x_i^k e^k) \right)$
O_2	$\sum_k \left(P^k - \left(\sum_{i \in I_-^k} x_i^k \right) b_{\hat{i}(k)}^k + \sum_{i \in I_+^k} x_i^k \left(b_i^k + e^k \right) \right)$
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 capsule g_k is placed on machine i

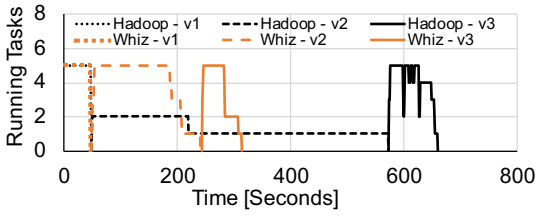
Parameters:

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

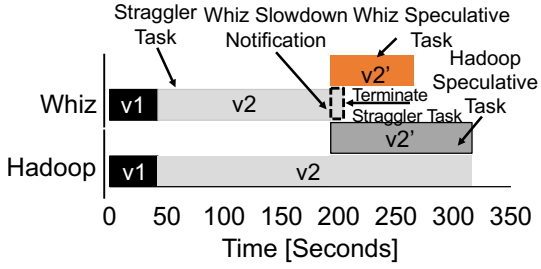
Table 7: Binary ILP formulation for capsule placement.

from v_2's unprocessed capsule. v_2 finishes processing the first capsule at 202s; v_2' processes the other capsule and finishes 1.7× faster than v_2' in CC.

Modification Predicates: 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



(a)



(b)

Figure 13: (a) Controlling task parallelism significantly improves WHIZ's performance over CC. (b) Straggler mitigation with WHIZ and CC.

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, WHIZ uses modification predicates for v_3 as follows - (a) if all the # entries of all keys in the capsule is > 100, then we unnecessarily don't launch a task; (b) otherwise we launch the task to do the sort. We observe that WHIZ ignores processing two capsules at runtime, and 6 tasks of v_3 (instead of 8) are executed; JCT is 165s (1.4× better).