A formalism for data processing systems

Timothy Hunter

December 27, 2017

Contents

This article draws on the foundations of measure theory and group theory to offer a formalism of large scale data systems. Starting from first principles, we show that most operations on datasets can be reduced to a small algebra of primitive functions that are well suited for optimizations and for composition into much larger systems, and in handling streaming and statistical data. We hope that these principles can inform the implementation of future distributed data systems.

1 Introduction

The explosion in the collection of data has lead to the rise of diverse frameworks to store and manipulate this data accordingly. These frameworks tend to either adapt some old paradigms or to emerge from the operational aspects of big data.

As far as old paradigms are concerned, the SQL language has enjoyed considerable success as a standard querying interface for the non-programmers. On the other end of the spectrum of complexity and expressivity, engineers in charge of processing large amounts of data have devised a number of tools to distribute computations against these large datasets, and have formalized these tools into operational abstractions such as MapReduce (ex: Spark, Flink, BigQuery).

At their respective levels, these abstractions share the idea of separating storage from compute. They free programmers from considerations around how to access and store data, and entrust the implementing frameworks with taking reasonable decisions. While this approach is effective in most scenarios, some specific domains such as machine learning or genomics rely on specific access patterns and hardware to enjoy good performance. For these domains, the separation between storage and compute breaks down: genomics and deep learning are concerned with streaming massive volumes of data through a handful of dedicated, powerful computing cores. Furthermore, in reinforcement learning, reactivity (and latency) is paramount, at the expense of overall computing throughput. Such problems have been successfully tackled by specialized frameworks (tensorflow, Ray, etc.) which do not presume a clean separation between storage and compute, but rather the capacity to access data in a specific pattern (streaming data for example). This paper, rather than assuming a strict separation between storage and compute, formalizes APIs in which the distribution and the scheduling of computations *adopts the structure of the data*.

If one looks at the landscape of the data processing systems, several trends emerge:

- unification of different paradigms (streaming, batch processing, interactive analytics)

- separating storage from compute does not work for workloads with high operational intensity like machine learning (see TPUs, GPU processing, John Canny's research).
- big data systems are used as foundations for implementing much larger systems.
- More generally, the same set of challenges that led to handling files with version control systems are now prevalent in continuous systems (TODO: cite matei)

When a system is designed to serve as the substrate on top of which more complex systems are built, establishing sound principles and structures is paramount. Without these, complex and unforeseen interactions emerge that expose some behavior that on the face of it, is a logical consequences of the initial specifications, but yields surprising or confusing results to the user of such a system. As such, this paper is mostly concerned with the soundness of its interfaces and delegates practical considerations to further study, with the hope that these interfaces are representative of current and future needs.

Artificial intelligence is not well captured by current data systems because of its roots in statistics, and its peculiar characteristics. Statistical workloads can usually trade some exactness in results for large computing gains, as shown by BlinkDB in some workloads. Current exploration has so far focused on starting from a high level goal (embedding constraints in SQL for example) and working these constraints through an existing engine. This paper proposes a bottom-up approach. It focuses on establishing a small, highly structured set of primitive operations that capture well the requirements of statistical techniques such as the bootstrap or model learning. In the streaming section, we will show that a statistical workload has the same characteristics as a streaming workload, and that for both we can establish some convergence and soundness guarantees.

Data processing systems are primarily focused on transforming data, and as such reading and writing data is a critical, but poorly formalized, part of such systems. For example, from the perspective of a programmer, to a distributed file system is often considered with the same flexibility as writing to a local file, even though the possibility - and the cost - of an error is considerably higher. This is why databases have been developed as data storage systems with strong guarantees (TODO: unclear sentence). An important motivation of this article is to provide a framework in which reading and writing data is considered a primitive operation, and on which all sorts of mechanical transforms, checks and optimizations can be applied.

This article is structured as such:

- it first introduces high-level principles based on observability of data (a prerequisite in distributed systems)
- it establishes necesary mathematical structure that naturally derives from these first principles

- it discusses possible ways to address temporal aspects and streaming data.
- it offers some hints at implementation and a proof of concept built on top of Spark, written in the Haskell and Python programming languages.

The code examples are done using mathematical notations and Haskell functions, which should be clear enough in most of the case.

2 General principles

This section is concerned with some ideas that hold no matter the type of system:

0. Perspective of the observer: only reductions can be observed.
1. The data representation is not observable
2. The flow of computations follows the structure of the data.
3. Reductions are deterministic
4. Failures are not observables

This paper starts from a few general principles and shows how can reconstruct some commonly used constructs in big data systems.

Observer perspective A dataset is a collection of values. Two operations can be done on a dataset:

- transforming it into another dataset.
- condensing the dataset into a single value. This is also what is called *reducing* the dataset into an observable (a value).

The observer principle states that the user (the observer) only has access to observables, and not to the dataset itself. This idea is familiar to physicists, who have long considered adopted a similar posture in quantum physics, in which the measure of the physical system itself (a wave function) is integral part of the design.

Abstraction of data representation In this setting, the data structure that backs the dataset is not observable and has no incidence on the results (the values of the observables). This is not respected by current systems: for example in SQL, the order of the results is not specified. More insidious, in Spark and Flink, results depend on the partitioning scheme of the data, sometimes with confusing results. This principle guarantees that making observations against a dataset yield a unique and well-defined value in all cases. (TODO: this is up to numerical precision issues). There are a variety of data structures to represent a collection of values, backed by an equally large number of implementations (B-Trees, sequences, lists, lists of lists, maps, etc.). This principle ensures that implementation details do not percolate through the interface, and we will see a few implementations in Section TODO. An additional motivation is the ability

to tie some formal statistical concepts such as distributions to justify and explain some choices.

The next principle is specific to data systems, and probably the most restrictive in terms of the structure that it imposes on the computation model. Yet, it covers a very wide variety of reductions that users would consider on data systems.

Distributed computations The flow of computations follows the structure of the data.

This naturally follows from the abstraction of the data representation: in the trade-off between separating compute from storage, or tying compute and storage, operations performed on the data should not depend on specific choice. Computations can either be run close to the data or require data movement. That last principle ensures that the programming model is oblivious to this implementation choice.

Deterministic observables Given the same inputs of datasets and observables, any observable shall yield the same results. (TODO: up to numerical precision). This is a natural assumption, yet it is often not respected by implementations. In this formalism, because we will consider operations of reading ang writing as observables, this will lead to important restrictions on the kind of input and output operations that can be performed.

Failures are not observable It is common to experience failure in large, distributed systems. TODO: have a principle that says that trying to access the value of an observable can yield a bottom value (error, exception, failure, etc.). However, the changes to the environment caused by producing a bottom value are not observable. Practically, this means that repeatedly trying to write a dataset to a file (possibly distributed), however many failures occur in the process, will not be observed. This is not respected in current systems for example, when the append mode is used, or at great cost in performance. TODO: this is pretty weak and unjustified.

The rest of this section explores in more details how this formalism applies to abstract data representations.

2.1 Datasets

We consider operations over some sets of values, in which we want to apply the basic principles outlined above. All the operations perform either on values or sets of values, and data processing consists in writing some functions that transform sets into other sets, or into values. As mentioned before, here are a couple of assumptions that we will work on:

- the exact representation of the data is not observable. This is often not verified in such systems.

- the data is considered static. It does not change over time. This will be addressed in Section TODO.

We consider here a simple formalism: call $\mathcal{U}$ the set of all the values, called the *universe*. We assume that the universe is countable and separable. Subsets of the set of all values can be tagged with *types*. A type is essentially a subset of all the values. Types are mostly useful when writing programs and for clarity, not so much for our mathematical derivations.

Definition: dataset. A dataset is a *finite measure* over the set of values. $d : \mathcal{U} \to \mathbb{N}$ with the restriction:

$$\sum_{x \in \mathcal{U}} d\left(x\right) < \infty$$

In terms of types, a dataset is simple a function that counts how many times a value is observed:

```
type Dataset a = a -> Nat
```

Note Basic arithmetic over sets ensures that the function defined above is a measure in the sense of distributions (TODO: add link to definition of measures). Since we work with a countable and separable set, we do not need the technical conditions that would be found in more more complex normed spaces. A part of this work is dedicated to working out the sufficient conditions that would hold in all generality.

2.2 Basic operations over distributions

We now define a number of common operations that will be present throughout the rest of the paper.

Union The union of two datasets is a dataset:

$$(d_1 \cup d_2)(x) = d_1(x) + d_2(x)$$

```
union :: Dataset a -> Dataset a -> Dataset a
union d1 d2 = \x -> d1 x + d2 x
```

This operation is of course symmetric, associative, commutative and has a zero element (the empty set). This is the most important of all the general operations, because it conveys some useful structure over datasets (a monoid and a semi-group).

Dirac The dirac of a dataset is the extraction of a single value at a given point. The name `dirac` comes from the theory of measures, which itself comes from physics. As is customary, we denote it δ_a.

```
dirac :: a -> Dataset a
dirac x = \y -> if x == y then 1 else 0
```

One trivial consequence is the fact that a dataset is the (finite) sum of diracs. This fact will have its importance later.

To simplify the notations, we introduce the following shorthand notation:

$$n \cdot \{x\} = \{x\} \cup ((n-1) \cdot \{x\})$$
$$0 \cdot \{x\} = \{\}$$

Mass The mass μ of a dataset is the count of all its points. The name sounds related to physics and we will see the intuition behind it later.

$$\mu(x) = \sum_{x \in \mathcal{U}} d(x)$$

```
mass :: Dataset a -> Nat
```

2.3 Point transforms

With this structure, there is not much we can do so far, apart from querying the value of a dataset at various points. We are going to add many more operations now. One of the simplest and yet most effective ways to transform a dataset is to apply some operation to each of its points individually. We are going to characterize the transforms that are the most regular with respect to the Distribution principle: if a dataset is a union, we would like the transform to respect this union.

Definition: regular transform. A function `f :: Dataset a -> Dataset b` is said to be *regular* if the following applies:

$$\forall d_1, d_2 \in \mathbb{D},\ f(d_1 \cup d_2) = f(d_1) \cup f(d_2)$$

It turns out that this adds a lot of structure to all the possible point transforms; in particular the transforms have to be automorphisms over the dataset monoid.

Proposition: neutral element. The empty set is a neutral element:

```
f {} = {}
```

The following proposition justifies the programming interfaces of most distributed systems:

Proposition: characterization. All the point transforms can be uniquely characterized by their operations on elements. More precisely, by the point function:

```
point_f :: a -> [b]
```

(up to a permutation of the elements in the return list).

TODO: proof

The last proposition shows that regular transforms are fully described by transforming individual points, and that each point can only get mapped to a collection of other points. We can now fully qualify transforms about how many points they produce for each point: exactly one (mass-preserving), at most once (mass-shrinking) or arbitrary.

Definition: Mass-preserving transform. A mass-preserving transform obeys the following invariant:

$$\forall d \in \mathbb{D},\ \mu\left(f\left(d\right)\right) = \mu\left(d\right)$$

It is fully characterized by a function `point_f :: a -> b`. Conversely, any such function defines a mass-preserving transform. This is the functional `fmap` familiar to functional programmers:

```
map :: (a -> b) -> Dataset a -> Dataset b
```

Definition: Shrinking transform. A mass-shrinking transform (or shrinking transform in short) reduces the mass of a dataset:

$$\forall d \in \mathbb{D},\ \mu\left(f\left(d\right)\right) \leq \mu\left(d\right)$$

Similarly, it is fully characterized by its optional return on each point:

```
mapMaybe :: (a -> Maybe b) -> Dataset a -> Dataset b
```

Definition: k-regular transform. A regular transform is said to be k-regular if for all dataset d:

$$\mu\left(f\left(d\right)\right) \leq k\mu\left(d\right)$$

Note that k can always be chosen a natural integer, since the description of a transform is equivalent to the description of its effect on each data point (which produces a finite number of outputs).

Transforms that are bounded and regular are the most desirable from the perspective of a computing model: the computation model is highly predictable with respect to the distribution of computations. We will see how some common operations can be rewritten as bounded transform for more efficent processing in a distributed fashion.

2.4 Reductions

Reductions correspond to the action of taking a dataset and condensing its information into a single value. This is the second fundamental operation one can do on a dataset.

```
r :: Dataset a -> b
```

Again, we would like this transform to obey some distribution principle: reducing a dataset to a single value should be done independently from the structure of the dataset, and it should be run in parallel if the dataset is the union of subsets. This is expressed through the following definition:

Definition: monoidal reductions A reduction r is said to be *monoidal*, or *universal*, if there exists a function $+_r : \mathcal{D} \to \mathcal{D} \to \mathcal{D}$ that obeys the following distributivity rule:

$$r\left(d_1 \cup d_2\right) = r\left(d_1\right) +_r r\left(d_2\right)$$

A monoidal reduction can be interpreted as carrying over through computations of the underlying structure of the dataset. It turns out that the property above is quite strong and imposes a lot of constraints on the definition of $+_r$:

Proposition: Monoid structure The relation $\left(\mathrm{Im}\left(r\right), +_r\right)$ describes a monoid over $\mathrm{Im}\left(r\right)$. The neutral element of this monoid is $r(\{\})$.

- **Characterization** This morphism is fully described by the values of `r` over diracs and `{}`.
- **Unicity** This monoidal law is unique over $\mathrm{Im}\left(r\right)$

Composition This trivial proposition has important consequences for optimizations: If `r` and `s` are both monoidal reductions, then the pair reduction: `\(x,y) -> (r x, s y)` is also a monoidal reduction. This allows arbitrary merging of various reductions against the same dataset into a single reduction, whil preserving the semantics of the program. We will see more examples of this proposition in action in section TODO.

How justified is this framework? It turns out that such computation generalize the notion of integration outside the ordinary ring of the reals. For any morphism between the set of distributions and some other monoid, one can define the integral as follows. Given a distribution d and a reduction r, the integral:

$$\int r\mathrm{d}d = \int_{\mathcal{U}} r\mathrm{d}\left(d\right) = \sum_x r\left(d\left(x\right) \cdot \{x\}\right)$$

One key intutive difference with the usual understanding of integration is that in the context of data manipulation, the emphasis is on the measure, not the reductor.

2.4.1 Reductions - examples

The following operations are examples of reductions.

count In this case, $+_{\mathrm{count}}$ is the regular addition over naturals.

sum $+_{\text{sum}}$ is the regular addition over reals.

max $+_{\max}$ is the pairwise maximum over 2 values.

indicator function $+_{\mathbf{1}}$ is the `OR` binary function.

top-K Given an order on a dataset, computing the top K values with respcet to this ordering is a common operation. $+_{\text{top}} : A^k \times A^k \to A^k$ takes the top K values out of the union of 2 lists.

variance, higher order moments Using `sum`, `count` and basic arithmetic, all other higher order moments can be obtained. Numerically stable implementations can also be obtained in one pass but this is beyond the scope of this article.

Here are some less common operations, but that still shows the power of this abstract representation.

bloom filter Given 2 bloom filters with the same parameters, the monoidal reduction is the point-wise `OR` operation over each bit of the bloom filter.

sketches This is the same idea with sketches, by taking pointwise sums of values.

approximate quantiles The algorithms used to build quantiles rely on intermediate data structures that are much amenable to be distributed using monoidal reductions. These data structures can the be post-processed to obtain approximate quantiles.

Naive bayes models and more generally models that rely on sufficient statistics without iteration.

Hashing and signatures Commutative hash functions are also monoidal reductions. An example of such function is the modular exponentiation $f(x_1, x_2) = a^{x_1+x_2} \mod b$ for well chosen prime numbers a and b.

2.4.2 Reductions - counter examples

A few operations do not enjoy such strong properties however.

Machine learning algorithms, in all generality. These models usually rely on solving a convex optimization problem, and convexity does not carry over the union. TODO: check submodularity

Modes of distributions. This is intuitive: it relies on counting point wise, which cannot be done with a single summary.

2.4.3 Reductions - applications

How to build observables from reductions? As we will see, a large number of observables are built from reductions in the form $f(r_1(d), r_2(d), \cdots)$, in which the final function f is pretty simple. A classic example is of course the mean,

which is the sum divided by the count. Such a decomposition is very useful from the perspective of the computational model.

- it is common that datasets come in batches $d_1, d_2, \cdots$. With such a decomposition, it is trivial to compute the update, as it is simply the application the reduction to the last batch. In that sense, these distributivity laws are akin to derivation in real analysis. More on that later.
- for data that is ordered, and for which we want to perform an operation such as windowing, or cumulative sums, this structure can substantially improve algorithm design by providing an automatic scheme to share intermediate data, for example using buttefly schemes. TODO: this is speculative, but this article is dense enough as it is.

2.5 Joins

A join is a fundamental operation from the relational model. In its more general form, it has the following function signature:

```
join :: Dataset (k, a) -> Dataset (k, b) -> Dataset (k, Maybe a, Maybe b)
```

Some variants such as left, right, outer and inner joins can be derived from this signature when combined with filter. For the purpose of this discussion, we just want to note that joins enjoy a wide variety of structural invariants, which makes them particularly useful. For instance, the following distributivity law holds:

$$\text{join}(a_1 \cup a_2, b) = \text{join}(a_1, b) \cup \text{join}(a_2, b)$$

- left joins are mass-preserving in their first arguments
- right joins are mass-preserving in their second arguments
- inner joins are mass shrinking

TODO: it should be possible to define joins just based on the axioms above and on the distributivity laws, but this does not bring much to the discussion.

2.6 Groups

A group corresponds to the notion that data points can somehow be associated together. As a byproduct it allows to express transforms on potentially many datasets all at once.

Definition: group A group is a function from values to datasets:

```
type Group k v = k -> Dataset v
```

Of course, this is a very high-level definition, and we will see how it works out in practice. A practical implementation of a group using the formalism above may be done as follow, thanks to Currying: when we expand the types, a group is the type `k -> v -> Nat`, or equivalently through Currying: `(k, v) -> Nat`, which suggest an alternative representation for groups:

```
type Group k v = Dataset (k, v)
```

We will use one representation or the other according to the situation. This remark also justifies why a specific type for groups of groups is hardly found in practice, as they can be reduced to a group with a single composite key.

Reductions over groups. Groups are interesting thanks to the two operations below, that allow to represent a dataset as a partition of sub-sets, and to apply a reducing operation on each of these subsets.

The shuffle operations takes a reduction and applies it logically to each of the logical subsets of the group, to form a final dataset.

```
shuffle :: Reduce a b -> Group k a -> Dataset (k, b)
```

Its opposite operation, the grouping operation, takes a collection of values and a partition function, and applies this partitioning:

```
groupBy :: (v -> k) -> Dataset v -> Group k v
```

Considerable work has gone on efficient implementation of these operations, mostly in the context of databases, so we will not discuss here how these operations get implemented. Instead we will consider them as fundamental parts of the framework.

Distributivity of shuffle In the context of our framework, we note the following distributivity property of the shuffle, which has important applications in practice when datasets are obtained in an incremental fashion. Given a reducer r and two datasets a and b, the following holds:

$$\text{shuffle}\left(r, a \cup b\right) = \text{fmap}\left(+_r, \text{join}\left(\text{shuffle}\left(r, a\right), \text{shuffle}\left(r, b\right)\right)\right)$$

In other words, if one has already computed a shuffle for some dataset a, there is no need to keep this dataset anymore if new data appears, the output of the shuffle is enough. Furthermore, this transform is entirely mechanical, which means that from the perspective of the user, there is no need to implement complex schemes to deal with historical data if it has already been processed. TODO: make stronger.

2.7 Miscellanous operations.

Here are a few other operations that can naturally be added to the framework above.

TODO: remove these?

2.7.1 Filter

Filtering is the canonical example of a contraction.

```
filter :: (a -> Bool) -> Dataset a -> Dataset a
```

2.7.2 Ordered reductions

Given some ordering on some keys of type `a`, this operation returns all the elements, sorted.

```
orderedReduce :: Ord a => Dataset (a, b) -> [b]
```

2.7.3 Normalized representation

It is useful for some operations to have a compact representation of a dataset without duplicates. We call the *normalized representation of a dataset* the dataset of values and associated counts of these values. This is the representation that minimizes the mass of a dataset while preserving the entropy.

The implementation of this operation is straightforward:

```
normalize :: Dataset a -> Dataset (a, Nat)
normalize d = shuffle count (groupBy id d)
```

A normalized representation is very useful in the context of sorting values or adding some entropy to a dataset. Note that this representation can also be approximated by a sketch if the values themselves are not required but just their existence.

2.7.4 Indexing and counting

Using normalized representation, one can add indexing and counting operations to a dataset without having to add special operators and without having to define an ordering on the values. Given a normalized dataset, one can define the following `enumerate` that multiplies each value along with an index:

```
enumerate (parallelize [('a', 3)]) == parallelize [('a', 0), ('a', 1), ('a', 2)]
```

A naive implementation of this function is not very robust in practice to large indexes. We show in the appendix how this indexing operation can be performed using $\mathcal{O}(\log n)$ shuffles and k-regular transforms only. In essence, this function can be implemented in time that is nearly bounded to the size of the dataset and independent of the largest index values, all using more primitive operations.

TODO: it is possible to have a nice implementation so that `enumerate` distributes over unions of datasets (approximately, using sketches), and have all the guarantees above.

2.7.5 Random operations

It is common to associate a random value x_i to each value z_i of a dataset. What does random mean in that respect? One possible view is that the random value x_i is not correlated in a statistical sense to the original value z_i: knowing one does not convey information about the other. I argue here that there are two ways to construct such random values, either by building a carefully constructed suite of values independently from the values of the dataset (the classical way), or using one-way functions (TODO: proper names), such as cryptographic functions, based on the values of the dataset.

In the classic way of building random values, one uses a seeded series. This is what Spark does. TODO: explain more how it works. This approach is inconvenient in the context of distributed datasets, though, since any repartitioning of the dataset can lead to inconsistencies, or even worse, to different results.

I argue that a cryptographic scheme should be the default in distributed datasets. It stems from the following observation: if each of the value in a dataset is unique (distinct from all other values), then applying a one-way function κ will construct values that are uncorrelated from both the original values and the other random values. If a dataset d is max-entropic, then the dataset $(d, \kappa(d))$ has random values.

This scheme can be extended to non-distinct datasets, using the `enumerate` operation, presented above. Such an approach has the advantage of generating values in a way that respects all the dataset axioms: generating random values is not a special operation, but simply yet another dataset transform.

3 Time series and streaming

This section is concerned about datasets that grow in time. The source of this growth is usually one of the following:

- the dataset is revealed incrementally in time as new data becomes available. This is the classic case of streaming datasets.
- the dataset is explored in an increasing fashion, with the hope that processing a small fraction of the dataset will provide an approximate but still useful result. This is the model followed by BlinkDB and G-OLA.

We consider these two applications as different facets of the same context, in which one obtains increasing samples all generated from some underlying distribution.

This begs the question: can we determine whether the observation mechanism is going to converge towards some value that is related to the underlying distribution, and how sensitive is this result to our sampling? Consider the following simple query that would check the deviation from some normal quantiles:

```
df.count(df >= 2 * df.stddev()) / df.count()
```

This query brings some interesting problems about what it means when being performed online. Can we assert that the quantity is going to converge as more data is observed?

This section attempts to provide an answer to this question, as well as a general framework that would provide some guarantees that the following query does not converge:

```
hash(df)
```

but that this query would:

```
max(hash(df)) - min(hash(df))
```

Definition: Stream. A stream s is a growing series of dataset:

```
type Stream a = Nat -> Dataset a
```

such that $s_t \subseteq s_{t+1}$.

The intuition behind a stream is that of a collection that is observed incrementally. Because it is essentially a dataset, most of the results over datasets carry naturally over.

TODO: using category theory to define the results?

One of the main considerations when dealing with streams is about stability and convergence: are results obtained for one part of the stream still valid in the future? We introduce a notion of stability that is common in the context of measures and distribution. It stems from the idea that a stream is the incremental observation of a distribution.

Definition: stable stream: A stream s is stable if there exists an integrable function $g : \mathbb{U} \to \mathbb{R}$ and a constant $K > 0$ so that for all elements:

$$s_t(x) \leq \mu(s_t)\, g(x)$$

$$\forall x,\ \frac{s_t(x)}{\mu(s_t)} \text{ converges}$$

$$\mu(s_{t-1}) - \mu(s_t) \leq K$$

(And recall that g is integrable if:) $\mu(g) < \infty$

The last condition is rather technical and easy to implement in practice by breaking an update into multiple sub-updates.

Most of the well-behaved streams are stable. One example that is *not* stable is the indicator stream: $s_t = [1, t]$. No integrable function can bound this stream, even if it converges pointwise toward 0.

3.0.1 Distances in data space

TODO: introduce a notion of distance between 2 elements of the same type. It is natural for any element:

- edit distance for strings (or vector distance)
- edit distance for arrays and maps
- usual L1 distance for the rest

TODO: build a norm based on that.

3.1 convergence properties of streams

Because we work in non-numeric spaces such as bloom filters or lists of strings, defining the convergence of a monoid is a bit more technical.

3.1.1 Notion of convergence

TODO: introduce the notion of limit using the o() notation instead of usual limit. Use distance.

Definition: stable reduction and signature: A reduction r is stable if for any stable stream s, there exists a value $u_r(s)$ so that:

$$r\left(s_t\right) \sim r\left(\mu\left(s_t\right) \cdot \left\{u_r\left(s\right)\right\}\right)$$

This function is called the signature of the reduction: sig(r)(s). The intuition of a stable reduction is that in the limit, as more and more data gets to be observed, the output of a reduction looks as if the dataset had been collapsed on a single value. For example, in the case of `max`, one only needs the largest value from the stream, everything else can be discarded. This intution carries also to more exotic objects such as bloom filters or `max_hash` (which is intimately related to HyperLogLog and approximate counts).

TODO: prove that the signature is unique.

TODO: explain the intuition with convergence theorems in statistics.

Definition: expectation of a stream This is the signature.

3.1.2 Examples of stable reductions

A lot of the usual reductions are stable, which is the interest of this notion: max, mean, sum, count, approximate quantiles, bloom filters, top-K, moments, mode

Some counter-examples: the following are not stable: hash, which is intuitive.

3.1.3 Main theorem

TODO: this is pretty intuitive, but it should be formalized: if s is a stable stream and f is a regular data function, then f(s) converges.

4 Implementation notes: datasets as an applicative functor

It is common to think of operations on datasets as monadic, and indeed this is how they are represented usually (CITE spark). In this section, we propose an alternative semantic that is strictly more general than monads and that we believe provides strong benefits in practice, in terms of usability and performance. Based on the formalism above, a dataset is an example of applicative functor. While this nuance may seem subtle at first, it offers a lot of advantages in practice. In effect, it allows a strict separation of a *data program* against arbitrary sources from the runtime. At the end of this section, we will show that all that has been discussed above can be implemented on top of three different runtimes: Spark, a generic SQL database, and the Pandas framework.

4.1 Generalities: dataset, the applicative

Spark (or for that matter, the other map-reduce based frameworks) make a distinction between *transforms* and *actions*. This distinction is inspired by the difference between the pure (functional) transforms and the effects, which are captured in the runtime monad of the underlying system. We propose to *treat all the effects of a data system as observables*. In more pedantic terms, data processing can be expressed as a category of applicatives, instead of a category of monads. In a monad, the value retured by one computation can influence the choice or result of another. For exemple, one could check if a file exist before opening another file. With applicatives, the structure of computations (the sequencing of effects) is fixed.

Recall that an applicative functor `f` is defined by the three following laws:

1. `map :: (a -> b) -> f a -> f b`, the functional map.
2. `pure :: a -> f a`, which takes a value and brings it into the context

3. `concat :: f (a -> b) -> f a -> f b` which combines two elements already in a context.

The essence of applicative functors is that *the transformation on the data can be described independently from the data itself*. This is a crucial point for distributed systems, in which datasets can be huge. This approach is different from the current implementations, which explicitly or implictly rely on an outer context to perform imperative tasks such as writing files, etc. This manipulation of an outer context follows monadic laws, which are strictly less general than applicative laws. Take the example of the following Spark snippet, in which a dataset gets written and then some other operations are perfomed:

```
val data: Dataset[Row] = ...
data.write.parquet("mydata")
// One could check here that 'mydata' has been written
data.write.parquet("mydata")
```

This code performs two consecutive actions (writing the data twice). Because these actions are imperative in the Spark programming models, they cannot be reorganized. Arbitrary operations could be performed between the two calls, in the Spark interactive model. This is a well known limitation of the monadic semantics in computation models, in which effects cannot be optimized automatically, even though the programmer may have additional insights into the program.

The question of I/O will be dealt with in more depth in the next section, but consider as an example the following optimization that is already possible with applicative semantics:

```
val data: Dataset[Row] = ...
val theMin: Int = data.min
data.write("/mydata")
val theMax: Int = data.max
val theSpread: Int = theMax - theMin
```

This sort of code would trigger two successive jobs in Spark, because Spark cannot infer in all generality that no effect takes place between the calls. If the side effects such as I/O are modeled as observables, the code above can be written with some slight modifications as following (as before, the type annotations are here for the clarity of discussion and are not necessary in practice)

```
val data: Dataset[Int] = ???
val theMin: Observable[Int] = data.min
val theMax: Observable[Int] = data.write("/mydata").then(data.max)
val theSpread: Int = sparkContext.run(theMax - theMin)
```

This code gets translated by the engine as the following:

```
data.cache()
val resultObservable: Observable[Int] = sql("select max(*) - min(*) from data")
```

```
val theSpread: Int = sparkContext.run(resultObservable)
sparkContext.write(data, "/mydata")
data.uncache()
```

A number of operations have happened:

- the operation of writing the dataset has been identified has independent from the other reductions. In fact, some extra caching logic is automatically inserted if necessary.
- both reductions were subsumed into a single pass over the data. This can significantly speed up computations and is accomplished automatically, through elementary analysis of the compute graph.
- the final difference is pushed into the same query, which will trigger Spark's whole stage code generation.
- finally, getting the result needs to be explicitly called out to run. With the proper assistance of the interactive system, this is hardly an issue in practice.

4.2 The case of inputs and outputs

Inputs and outputs are the most important effects when datasets are concerned. As far ar our formalism is concerned, they are treated as an observable. Indeed, the signature of the writing function in Haskell is:

```
writeData :: WriteConfig -> HdfsPath -> Dataset a -> Observable ()
```

This gives the program a large amount of flexibility in reoarganizing the computations if necessary, and in providing a consistent way to deal with effects. Some current capabilites include automatic checks on permissions and overwriting. Furthermore, in the context of streaming data, the code can handle streaming data with no change. As mentioned above, if the program detects that a write is not necessary (such as a place being overwritten later but never read in between), it removes such a write and all the dependent operations.

As an interesting application, a number of seemingly illegal programs now make sense, owing to the uniform treatment of all side effects as observations. Consider the following trivial function that takes a dataset, writes it and then returns the count of observations:

```
def function(d: Dataset[a]): Observable[Long] = {
  d.write("/path").andThen(d.count)
}
```

This is an aggregation, and as such it can be used in groups:

```
val data: Dataset[(Int, a)] = ...
val data2: Dataset[Long] = data.group().reduce(function)
```

What happens when evaluating `data2`? A set of files gets created, following the grouping convention of the underlying implementation of the file system. In the case of Spark, the following files would get created:

```
/path/key=value1/record0000.parquet
/path/key=value1/record0001.parquet
...
/path/key=value2/record0000.parquet
...
```

More generally, inputs and outputs can be treated in a composable manner within programs: one can assemble functions that contain reads and writes in a way that is safe and composable. From experience, significant amounts of effort are spent in practice on understanding the data flow. Making data writes and reads a first class citizen provides much more visibility for the programmer and the runtime on the actions of the system.

As an other example, the runtime can checks some invariants, the departure of which are much maligned in practice. Consider the following (simplified code):

```
val data: Dataset[a] = ???
val o = data.write("/path")
val o2 = data.read("/path").count
context.run(o, o2)
```

This would be valid code in a monadic setting, but at the expense of not exploiting parallelism to the fullest extent. Because the two effects are clearly not commutative (a common situation for interesting applicatives), our current prototype will refuse to run this program and detect that some additional ordering must be introduced by the user.

4.2.1 Writing modes

Owing to the genearl philopophy of determinism, some operations have retstricted semantics compared to existing systems. In particular, there is no `append` mode when writing, as this would violate the principle that observables are deterministic given the same inputs. We propose an alternative though, using groups. As mentioned, groups are written in a partitioned way, based on the key of the dataset. It is common for new data to be associated to a separate key. Also, because of the strong associativity laws of the datasets, for common, simple operations, the framework can detect if the update is incremental, and only update the required parts of the files.

Writing presents interesting challenges when trying to use distributed file systems that offer fairly weak forms of consistency. This has been covered in other pices of literature, so we will not cover it here.

4.3 Miscellani

Some operations do not carry over with this scheme, for example when the schema depends on the data. This is the case in a few operations:

- pivot: the columns depend precisely on the value of the data, so arguably it does not work. In our paradigm though, there are a few ways to emulate this behavior.
- schema discovery with unstructured sources (for example JSON or CSV). It is common when starting with a new source to discover its schema. In that case there is indeed not much that can be done. In practice though, one usually separates the phase of complete discovery, in which nothing in known *a priori* about the dataset, and the phase of manipulation, in which some content of the data is at least expected (for example, we expect a feature columns with arrays of doubles). This is where a functional style can help by clearly delineating the expected schema from the other parts.

5 Appendix 1 - sampling and expansion algorithms

TODO: show how one can do sampling with and without replacement in bounded computation time.

TODO: show to to do efficient parallel bootstrapts with monoids

TODO: show how this merges into bag of little bootstraps