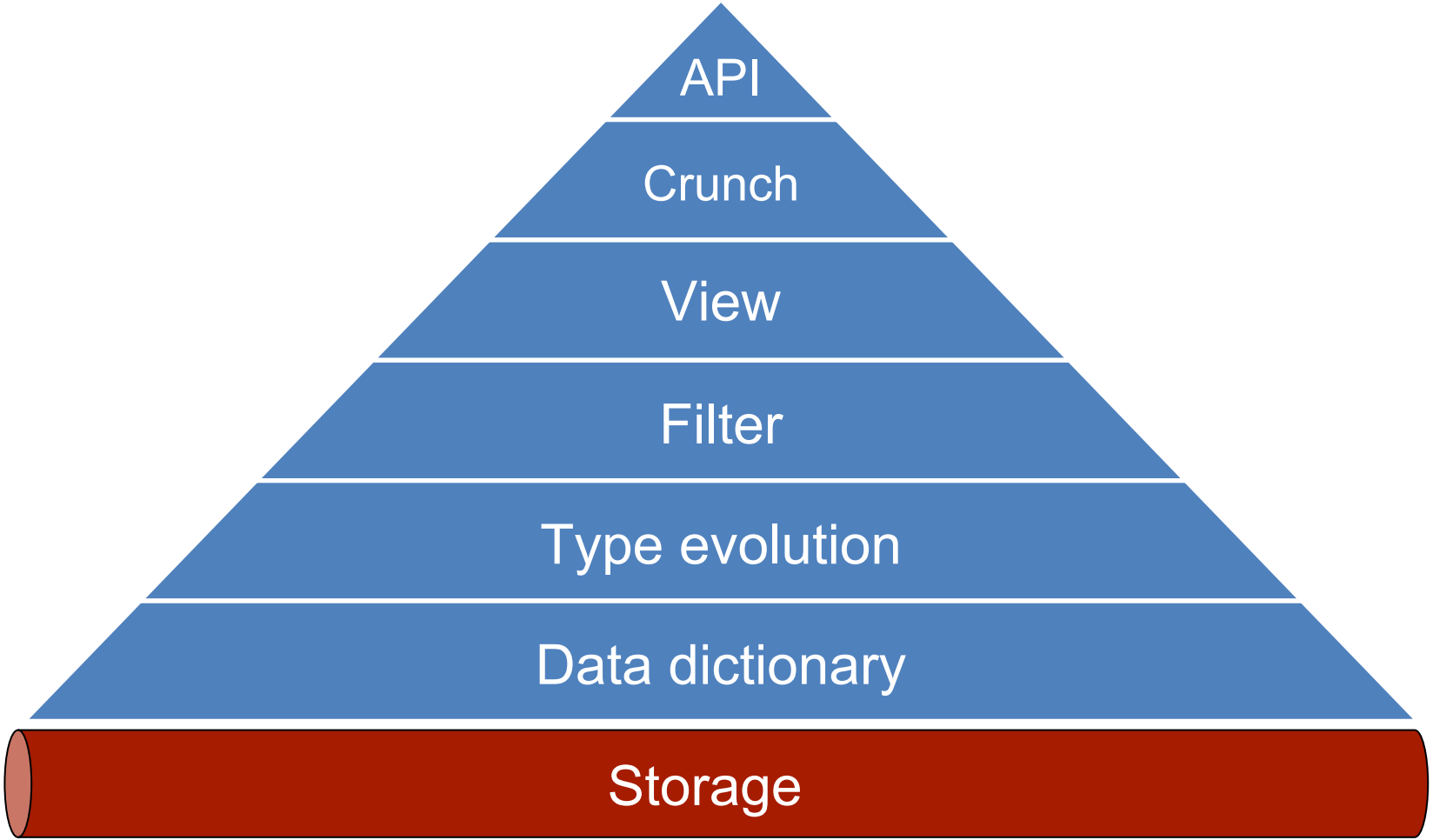
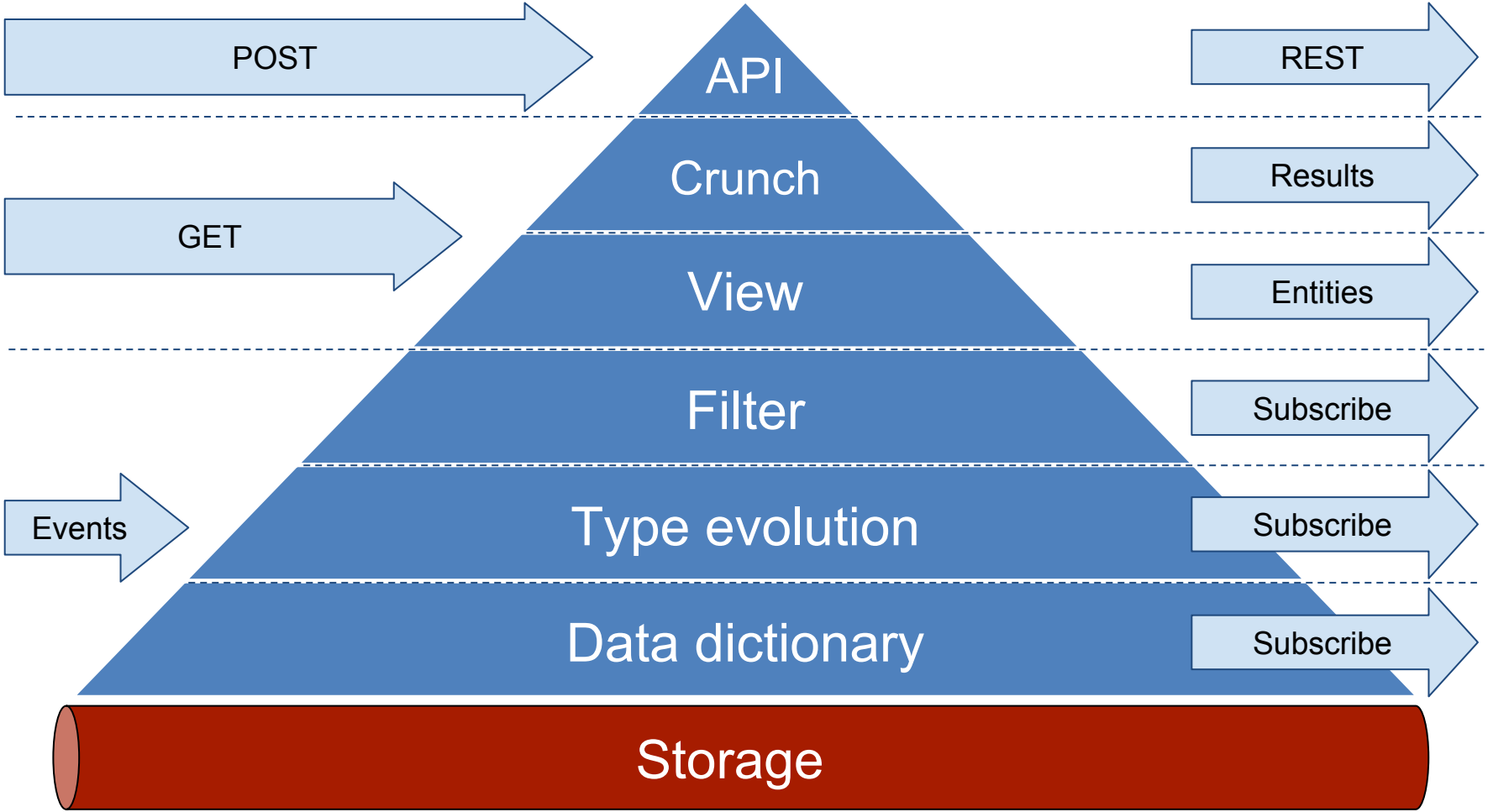
Current

Internal, 10/25/2015


API
Crunch
View
Filter
Type evolution
Data dictionary
Storage


POST
GET
Events
API
Crunch
View
Filter
Type evolution
Data dictionary
Storage
REST
Results
Entities
Subscribe
Subscribe
Subscribe

Gradual Introduction

Bet on Current’s strong sides. With no strings attached.

From the system of record only, to owning an end-to-end business logic domain.

Persistent Event Log

Define the data dictionary and persist events in a strongly consistent pub-sub storage.

Do:

- Define the schema of event types to persist
- Run your Current DB server

Get:

- Publish and subscribe
  - strongly respecting the schema
  - from embedded C++, to HTTP/curl and language bindings.

```
// Schema for the data to persist.
STRUCT(User) {
  FIELD(name, string);
  FIELD(age, int);
};
```

```
// Store as `db`, expose as `stream`.
PUBLISH(db, PUBLISH_TYPES(User));
SUBSCRIBE(db, stream, EXPORT(User)) {
  PASS(User);
};
```

```
# Publish, a single record.
$ curl -d '{"type":"User","fields":{"name":"Bob", age: 31}}' $URL/db

# Subscribe, get an infinite stream.
$ curl $URL/db/stream  # add parameters to tweak it.
{"type":"User","fields":{"name":"Alice", age: 27}}
{"type":"User","fields":{"name":"Bob", age: 31}}
^C
```

Type Evolution

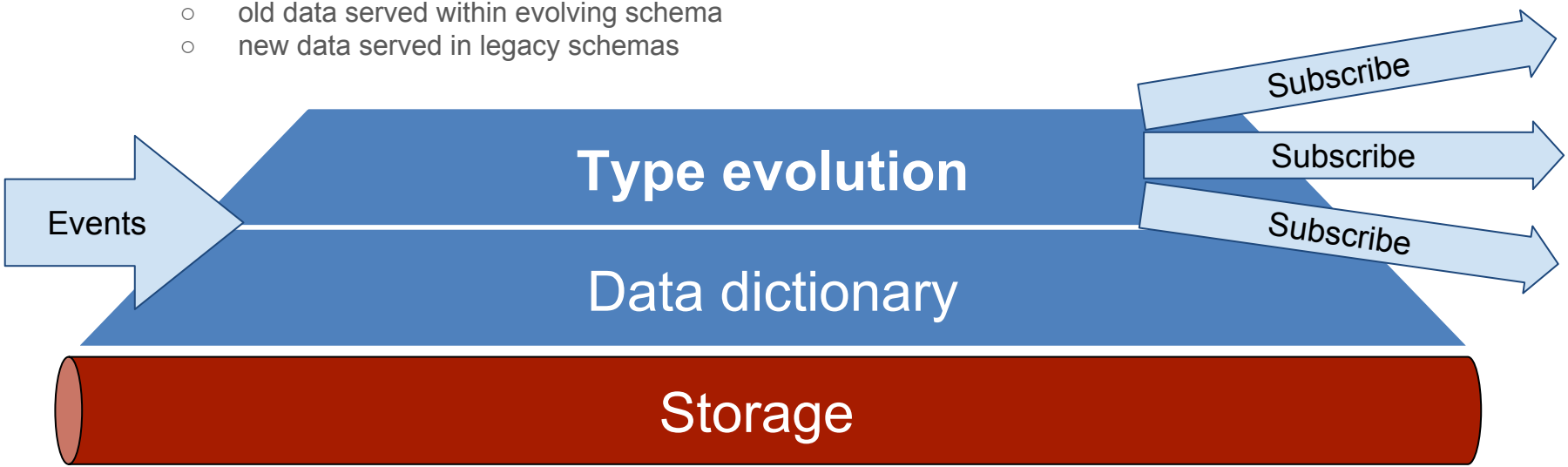
Grow the ontology of persisted event types.

Do:

- Create new versions of existing and introduce new types
- Respect existing data contracts and define new schemas

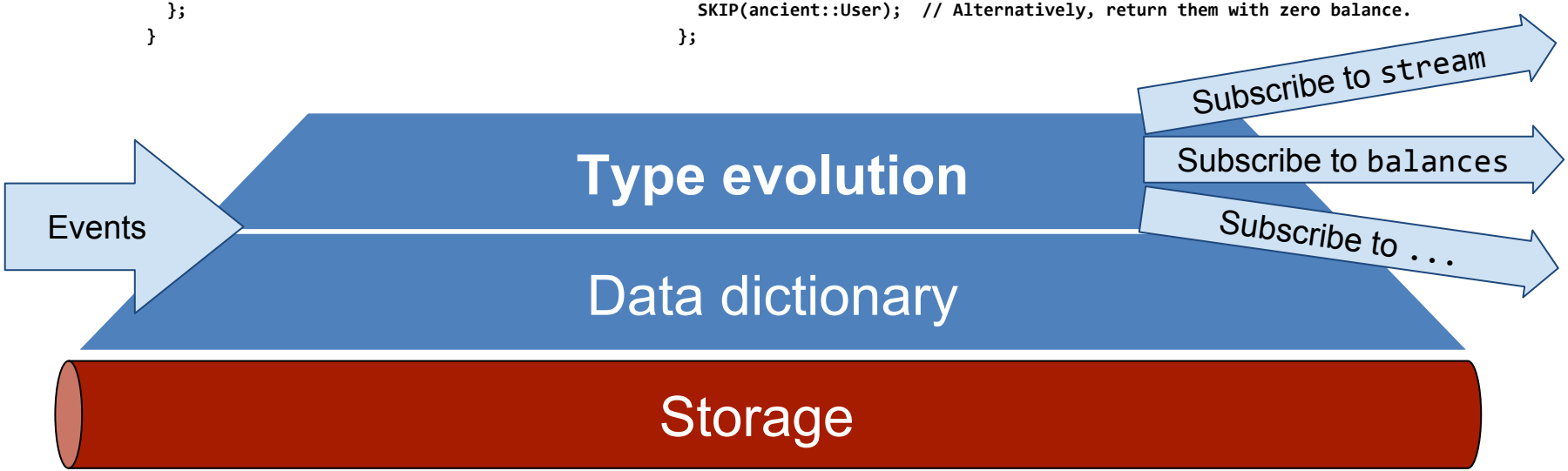
Get:

- Existing code is reused and stays backwards-compatible:
  - old data served within evolving schema
  - new data served in legacy schemas



```
// Introduce schema epochs.
namespace ancient {
  STRUCT(User) {
    FIELD(name, string);
    FIELD(age, int);
  };
}

namespace modern {
  STRUCT(User) {
    FIELD(name, string);
    FIELD(age, int);
    FIELD(balance, double);
  };
}
```

```
// Define the logic to represent one type as another.
VIEW_AS(ancient::User, modern::User) { ... }

// Respect newly added types within existing subscriptions.
PUBLISH(db, PUBLISH_TYPES(ancient::User, modern::User));
SUBSCRIBE(db, stream, EXPORT(ancient::User)) {
  PASS(ancient::User);
  PASS_AS(ancient::User, modern::User);   // Downcast.
};

// Add a new subscription under a different name.
SUBSCRIBE(db, balances, EXPORT(modern::User)) {
  PASS(modern::User);
  SKIP(ancient::User);  // Alternatively, return them with zero balance.
};
```



Custom Subscription

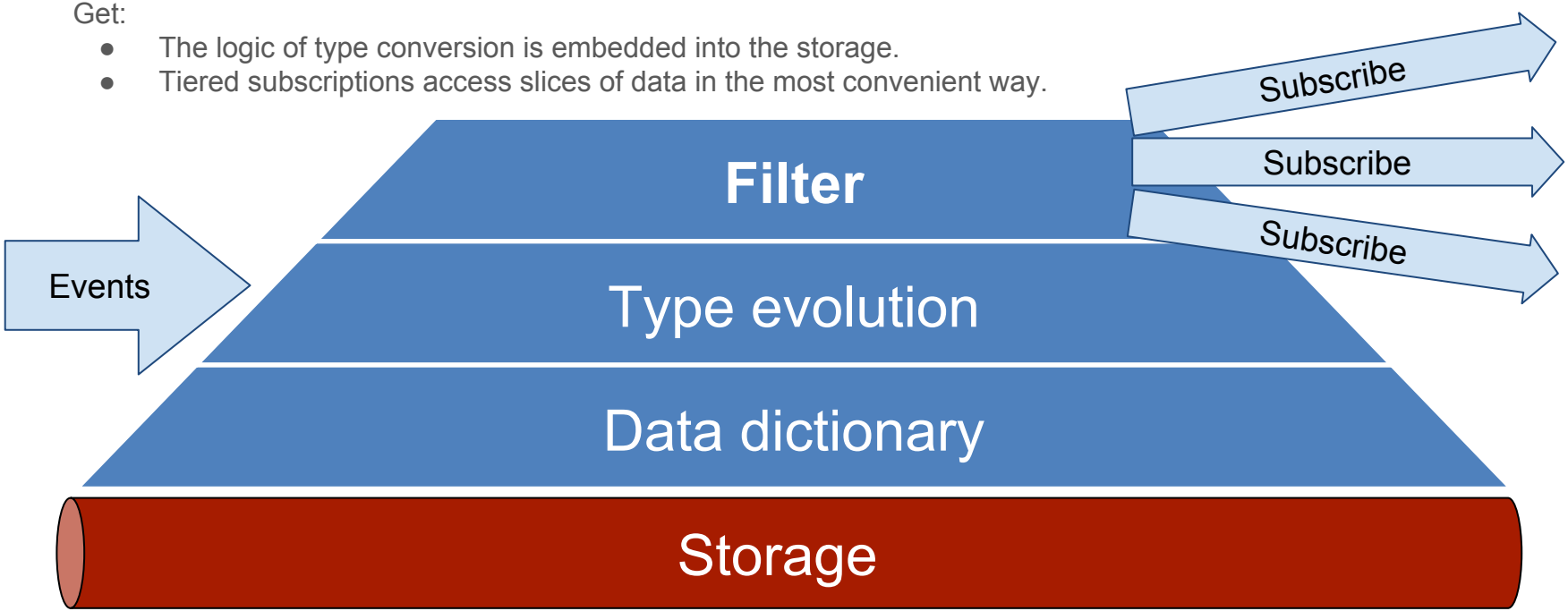
Embed logic to filter and transform the stream the user subscribes to.

Do:

- Introduce custom data transformation logic into subscription schemas.

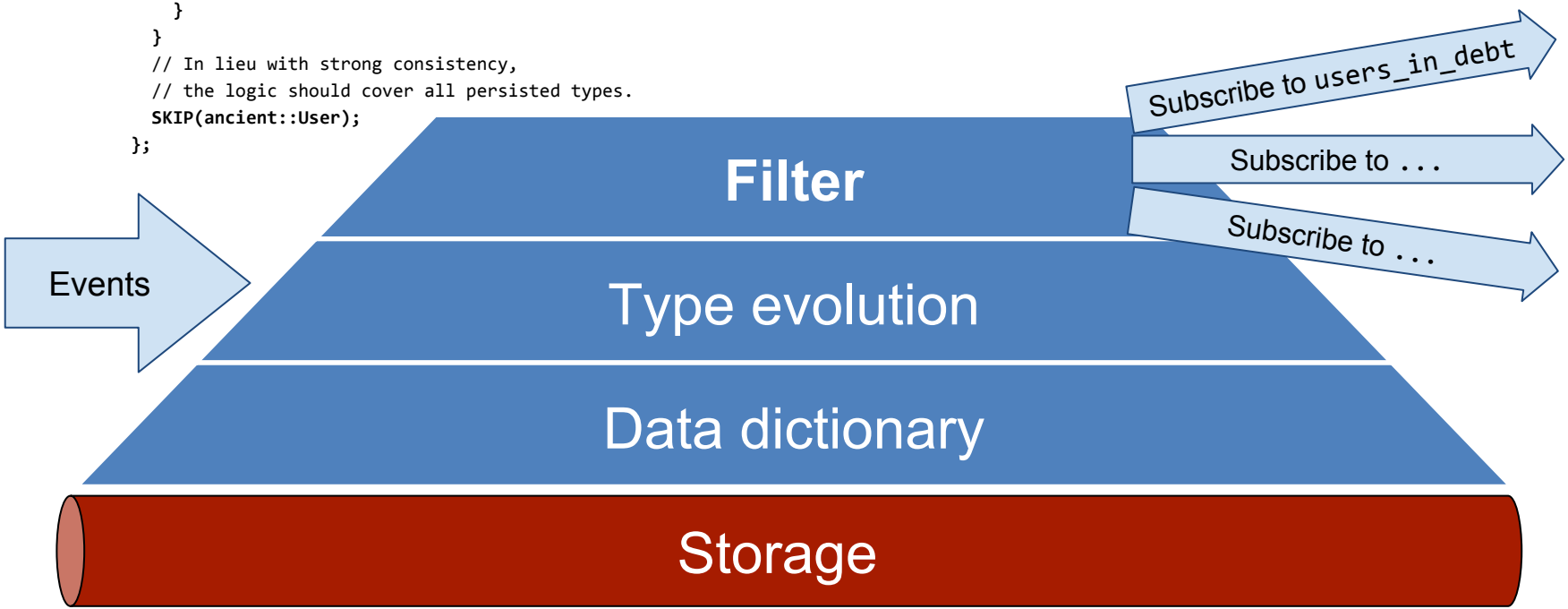
Get:

- The logic of type conversion is embedded into the storage.
- Tiered subscriptions access slices of data in the most convenient way.

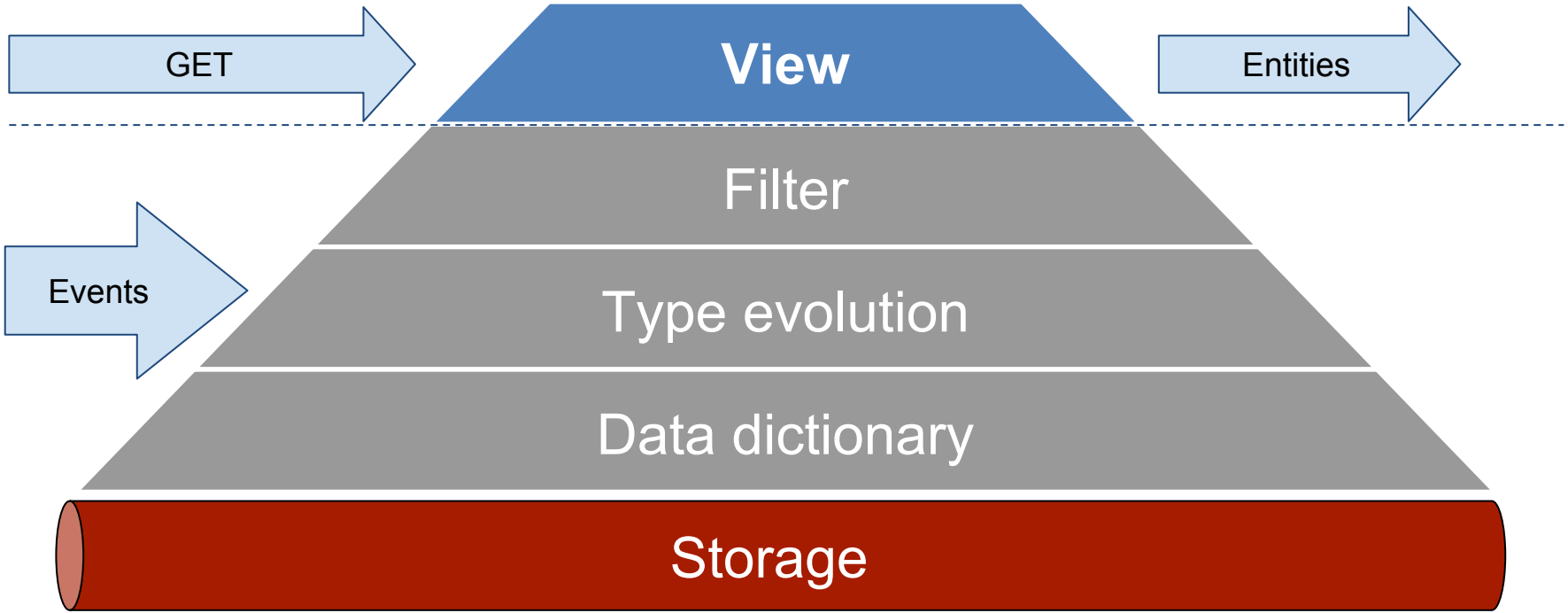


```
// Introduce a filtering subscription
// to the stream of users with negative balances.

// List all exported types.
SUBSCRIBE(db, users_in_debt, EXPORT(modern::User)) {
  // Conditionally pass some `modern::User`-s.
  ON(modern::User user) {
    if (user.balance < 0) {
      emit(user);
    }
  }
  // In lieu with strong consistency,
  // the logic should cover all persisted types.
  SKIP(ancient::User);
};
```

```
# Example.
$ curl $URL/db/users_in_debt
{“type”:”User”,”fields”:{“name”:”Donald”, age: 69, ...}}
^C
```



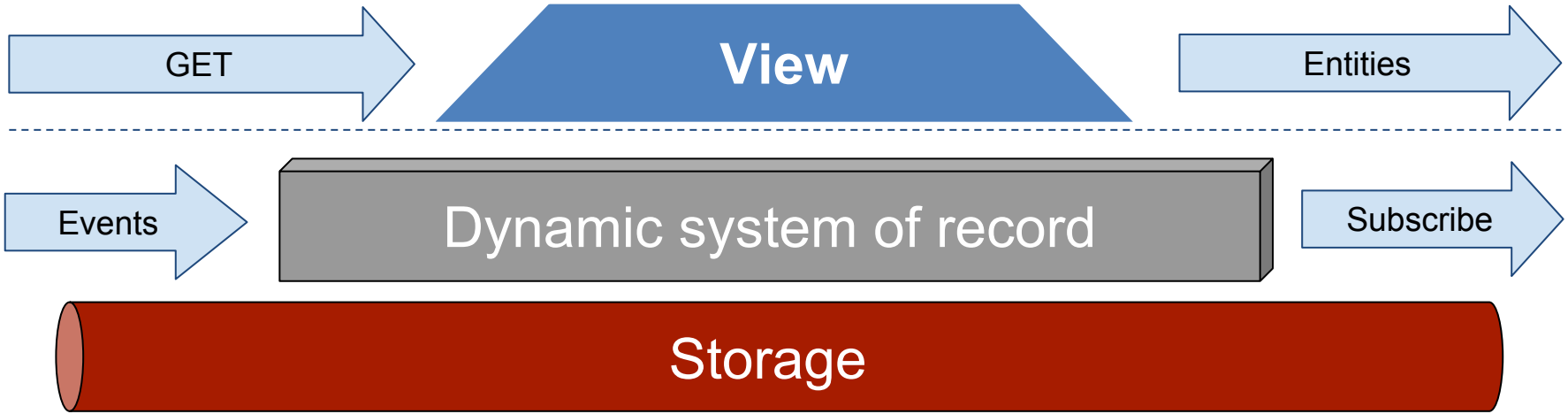
Materialized Views



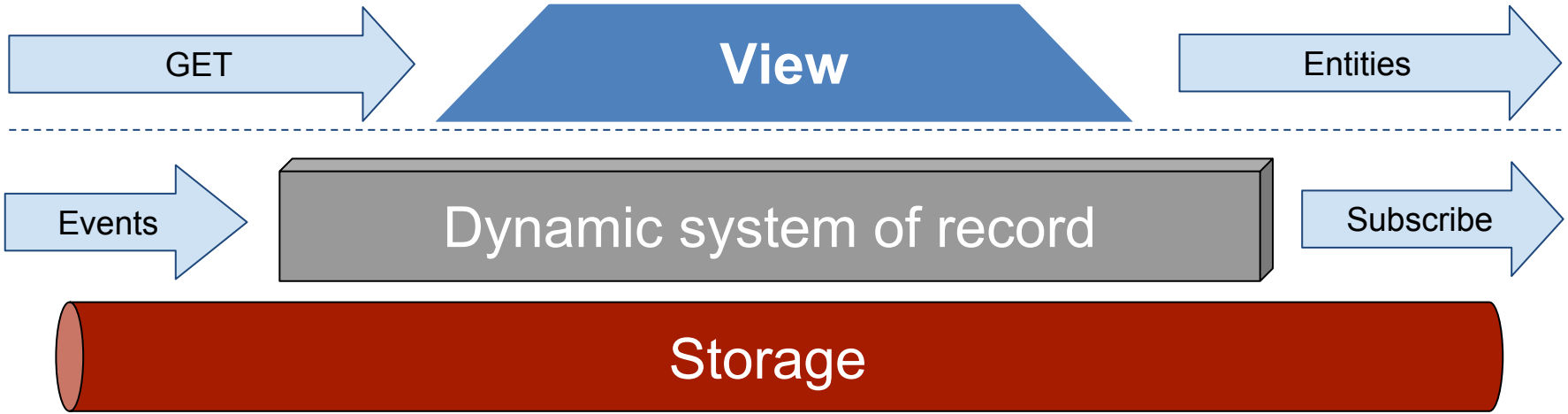
Materialized Views

Unite data dictionaries under a single system of record supporting materialized views.

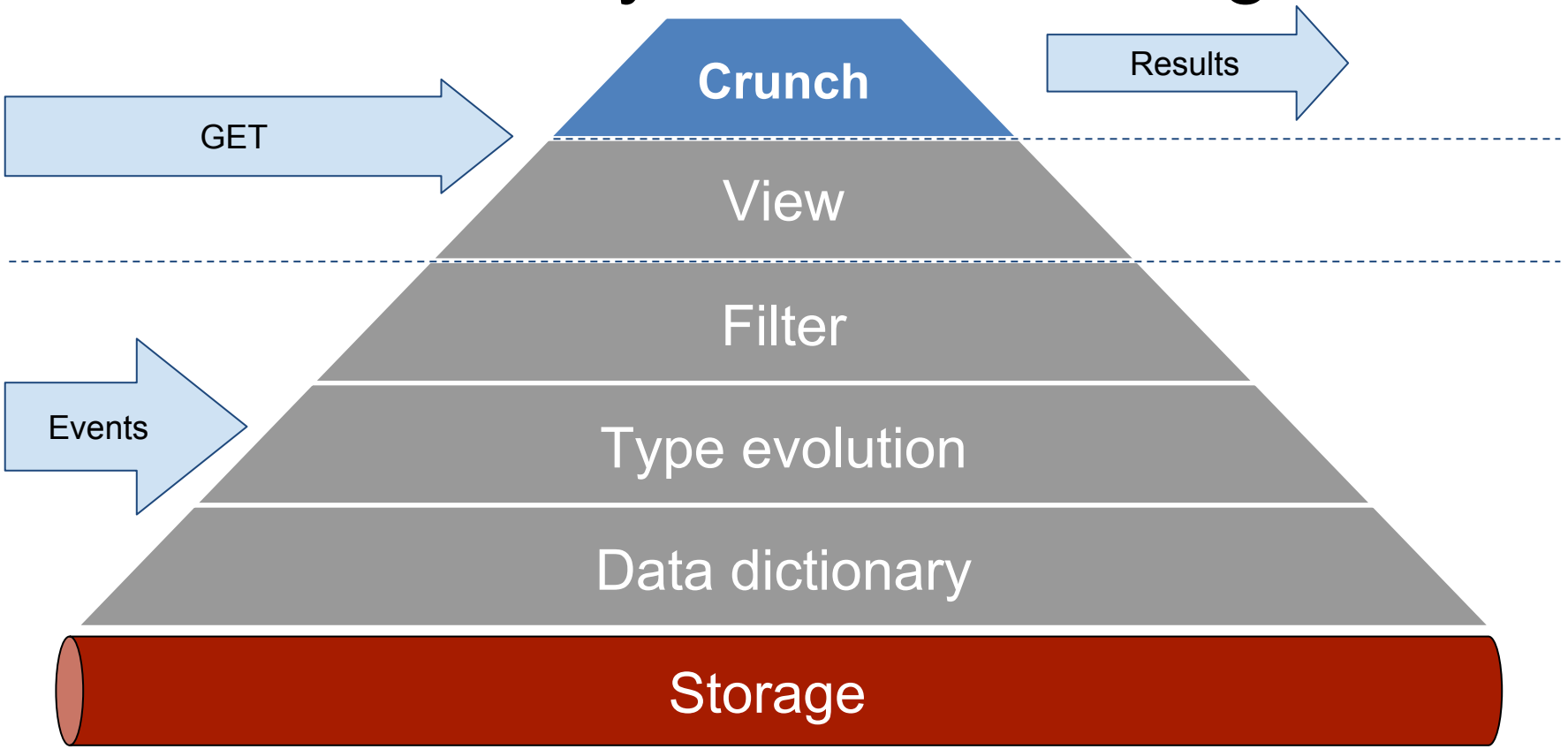
Introduce state to custom subscription logic.
Template types for basic view types, such as dictionary or top content by score.



```
VIEW(products) {
  // Persistent in-memory view.
  map<string, Product> products;

  // The logic to keep it up to date.
  ON(Product product) {
    products[product.id] = product;
  }
  ON(ProductNoLongerAvailable product) {
    products.erase(product.id);
  }
  // `product_id` is the URL parameter.
  HTTP_GET_ENDPOINT(product_id) {
    const auto result = products[product_id];
    if (Exists(result)) {
      SEND_RESPONSE(HTTP.OK, Value(result));
    } else {
      SEND_RESPONSE(HTTP.NotFound, "Product not found.");
    }
  }
};

$ curl $URL/db/products?product_id=...
{"type":"Product",...}
```



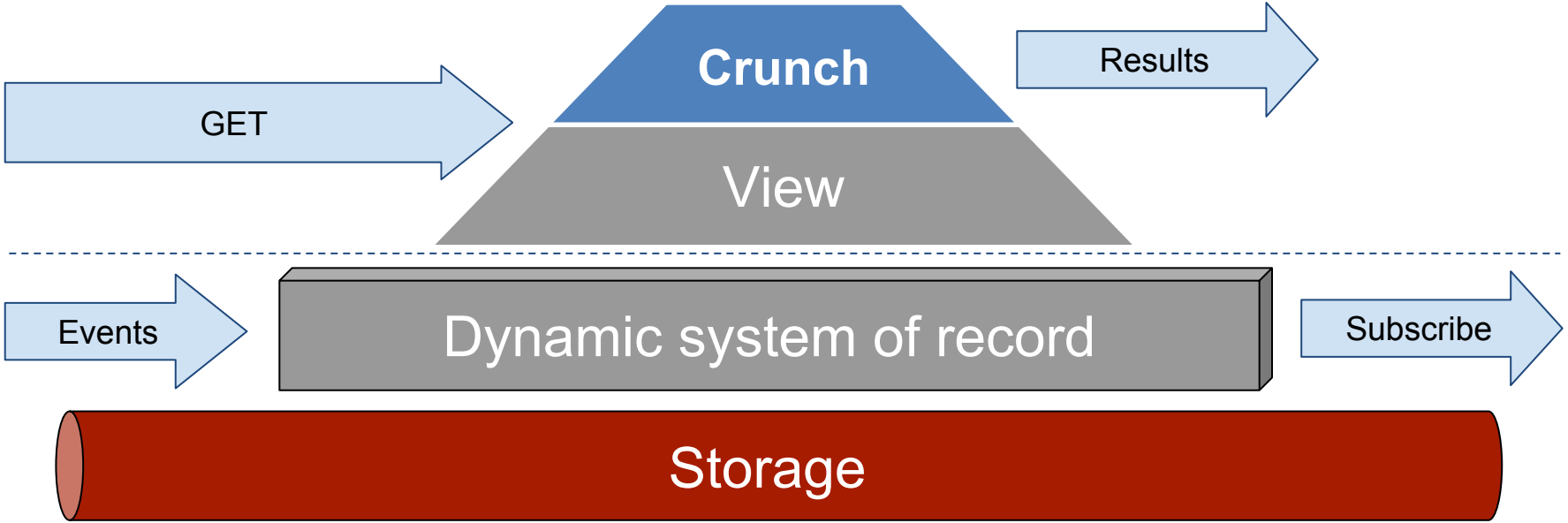
On-the-fly Data Crunching



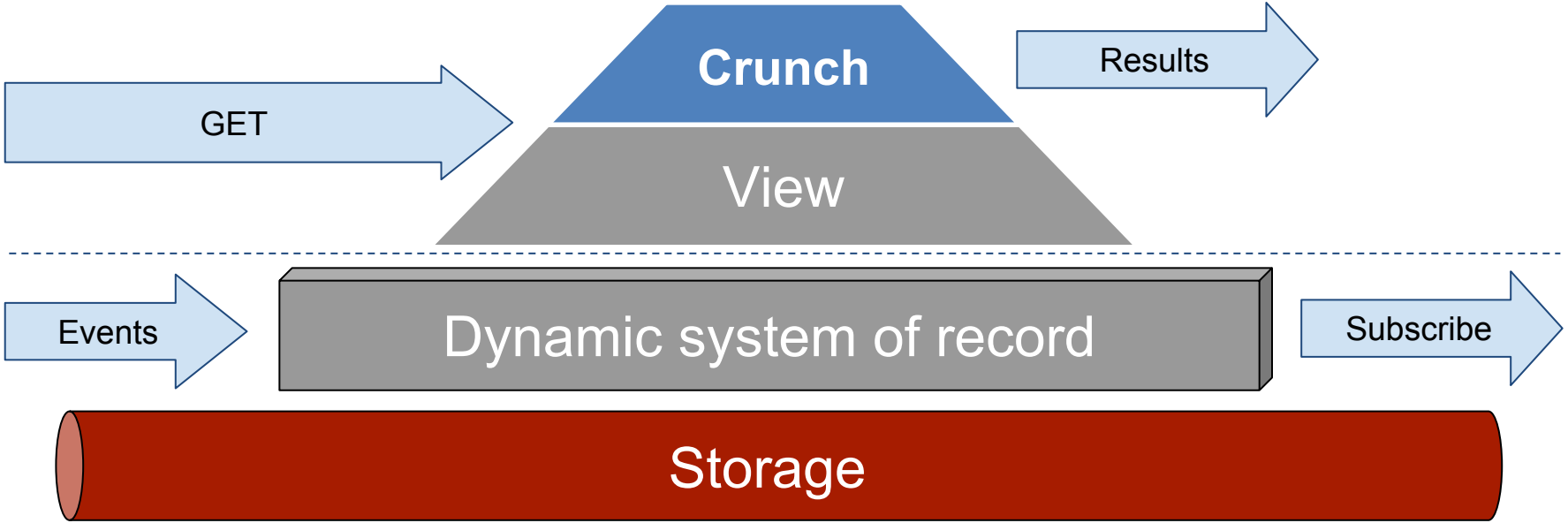
On-the-fly Data Crunching

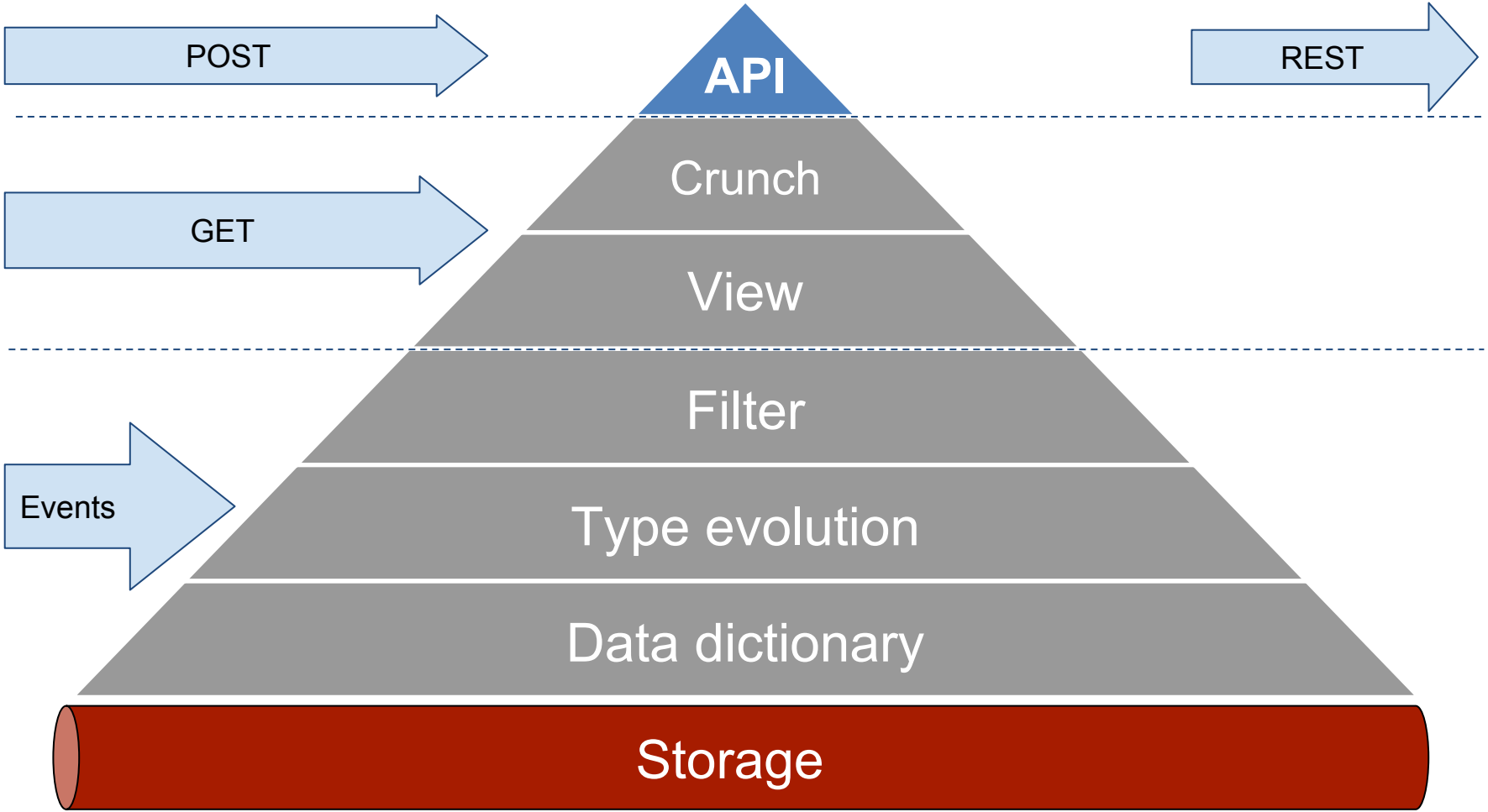
Same logic that maintains materialized views can crunch incoming data and publish back into the persisted log.

What is published back can be subscribed to, used by other materialized views, crunchers, and the API.



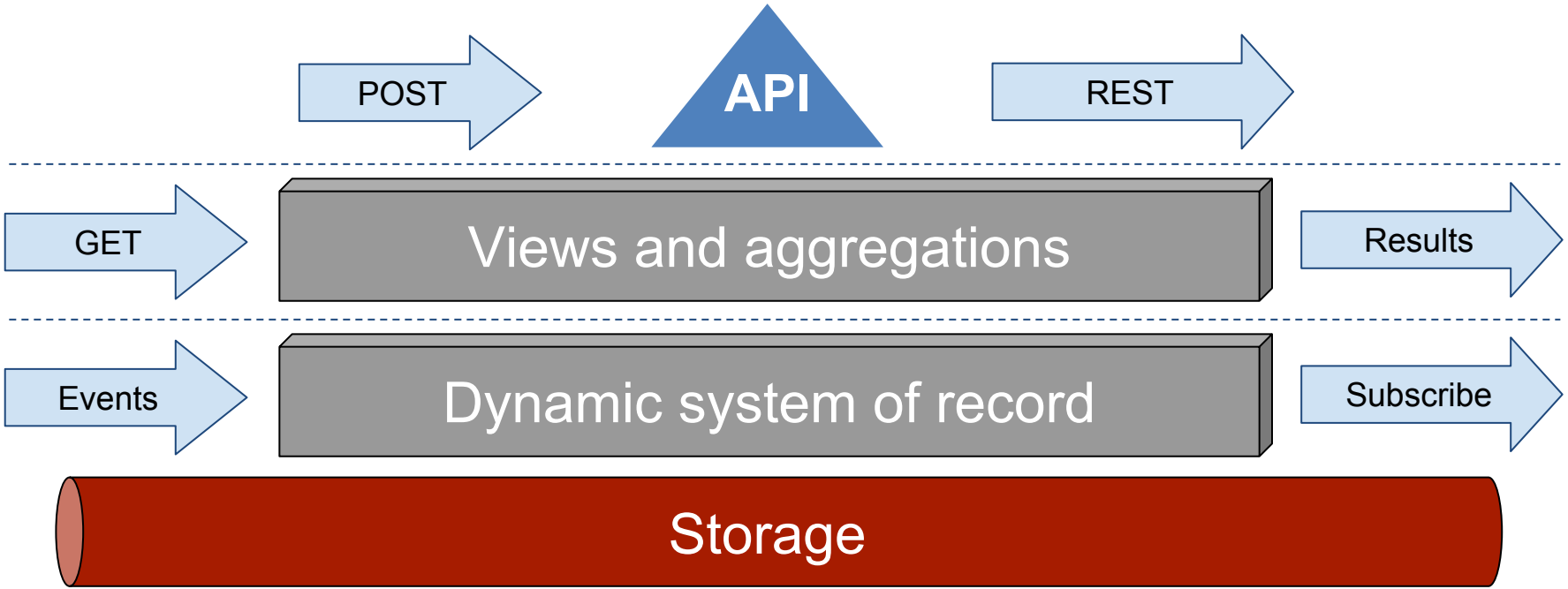
```
// Persisted type.
STRUCT(TopProductsSnapshot) {
  FIELD(products, vector<Product>);
  static BuildFromCounters(...) { ... }
};

// Crunching and publishing logic.
CRUNCHER(top_products) {
  CRON(EVERY(HOUR(1)));
  map<string, int> product_checkouts;

  ON(Checkout cart) {
    for (const auto& item : cart.items) {
      ++product_checkouts[item.product_id];
    }
  }
  ON(Cron) {
    publish(TopProductsSnapshot::BuildFromCounters(product_checkouts));
    product_checkouts.clear();
  }
};
```




POST
API
REST
Crunch
GET
View
Filter
Events
Type evolution
Data dictionary
Storage

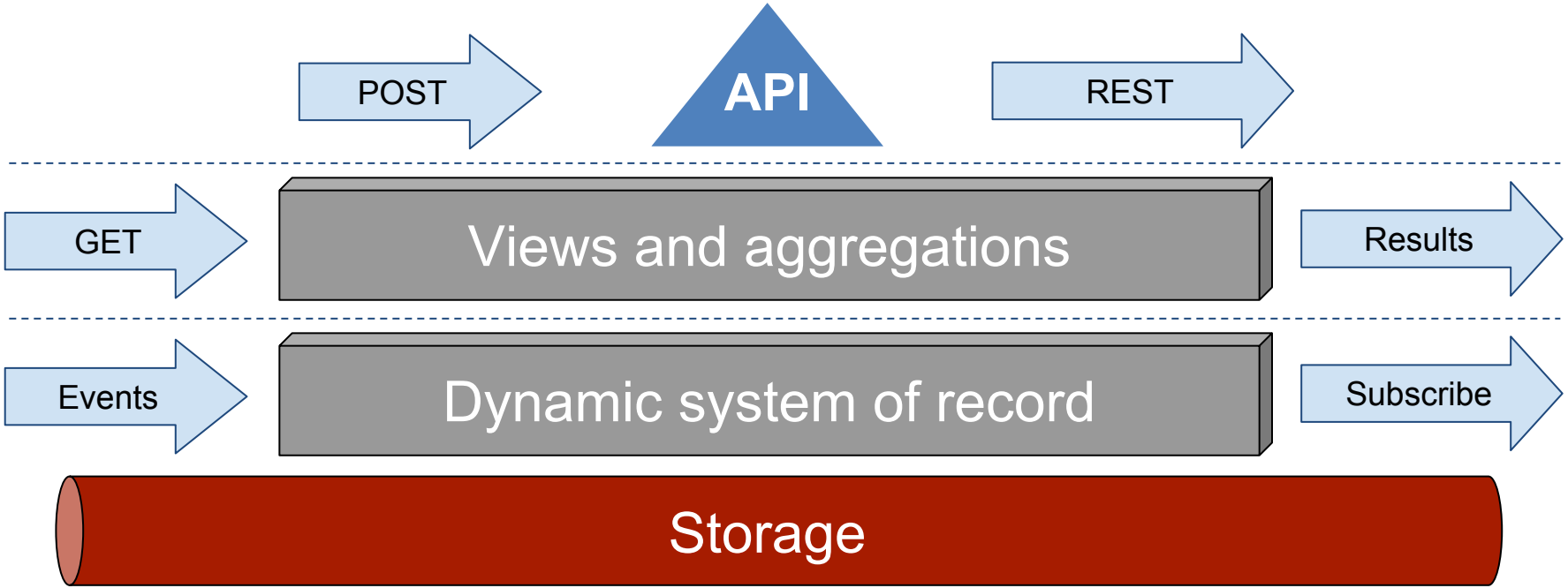
API & Embedded Business Logic

Migrate your most data-heavy domains into Current.



API & Embedded Business Logic

Example: Real-time recommendations of top products to appear on the main page as promotions, and as search suggestions.



2015: System of record, persistence, publish-subscribe.

2016: Views, data crunchers, API.

Thank you