Go + Cassandra: Insertion and Retrieval Patterns

Srdjan Marinovic
@a_little_srdjan

About me / Disclaimer

- Former CS researcher in Security and Distributed Systems.
- Currently building IoT cloud services at wirelessregistry.com
  - #IoT #BigData #Privacy

- I am neither a C* nor a Go jedi :)
- The following ideas have nevertheless yielded working solutions.
  - And I keep on learning daily about both techs.

Goal for this talk

- Share my account of using Go for concurrent insertion and retrieval operations against Cassandra.
- Can be a starting point in linking Go APIs with Cassandra backend.
- Helps me solidify my own knowledge, and produce some internal documentation :)

Contents

- Background
  - Cassandra DB, Go concurrency
- The Simple Worker Pool pattern (basic building block)
- INSERTs
- SELECTs

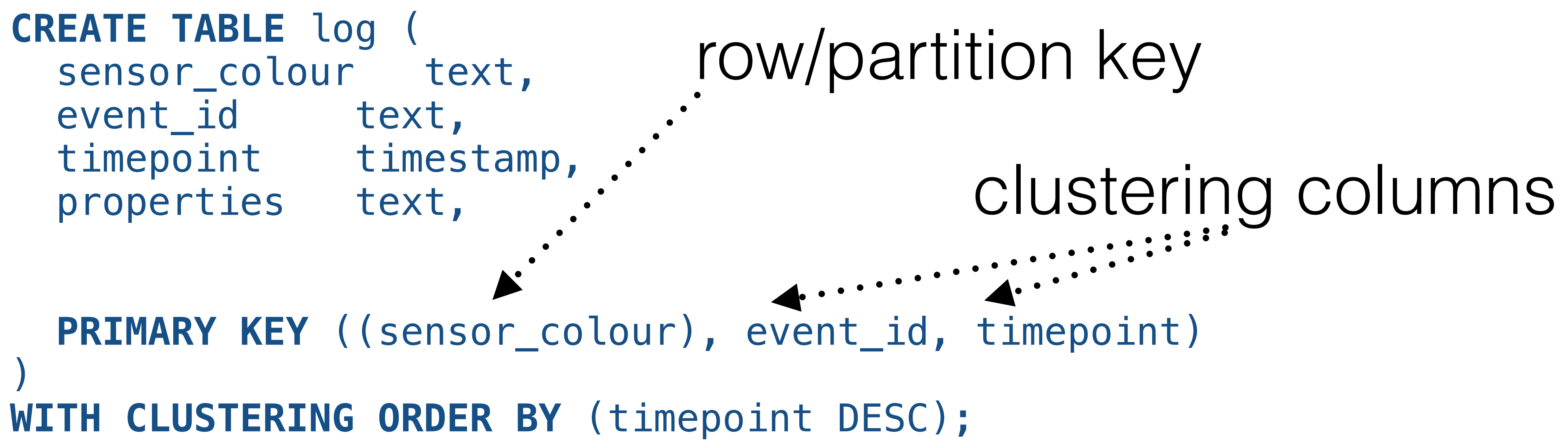
- **Note 1**. This is a relatively short and a light-weight talk.
  I will highlight pointers to the rabbit holes.
- **Note 2**. This is not a talk about configuring gocql (set-up clusters, sessions).
  See the *Cassandra and Go* presentation by Al Tobey.

C* overview

- A column-family store (ala BigTable and HBase).
  - Has a very nimble query language CQL.
- Tunable replication and read/write consistency properties.
  - I.e. you can tune the C+A (following the CAP theorem).
- Peer-to-peer distribution model.
  - All nodes are equal, no single point of failure.

C* data model

- Hello World example = “Collect a stream of events from sensors”.



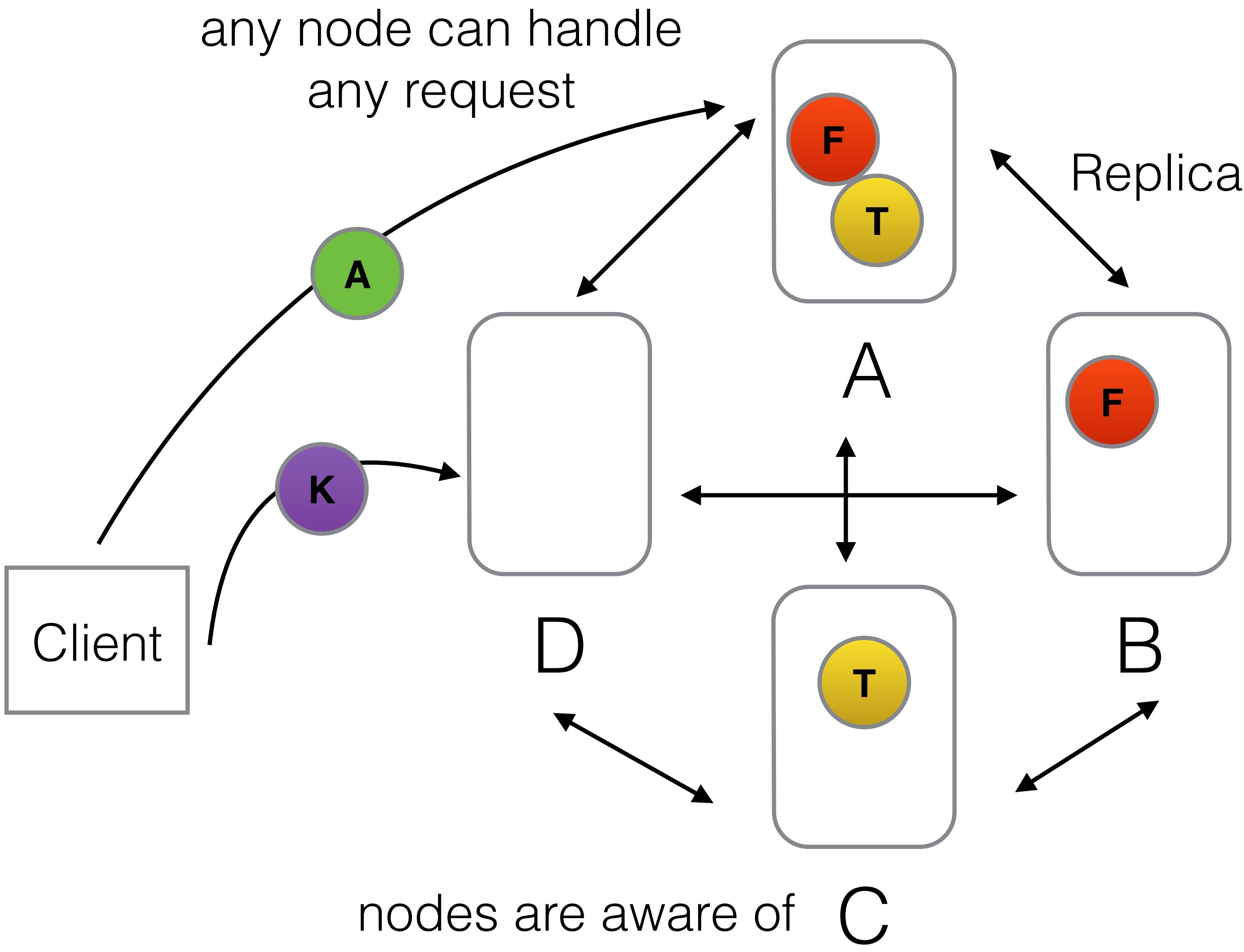
```
CREATE TABLE log (
  sensor_colour   text,
  event_id        text,
  timepoint       timestamp,
  properties      text,


  PRIMARY KEY ((sensor_colour), event_id, timepoint)
)
WITH CLUSTERING ORDER BY (timepoint DESC);
```

row_key=**sensor_green**	col_name=**ID1:12132**:*properties*	col_name=**ID2:12131**:*properties*
	col_value=**on**	col_value=**off**

. . .

C* distribution model



any node can handle
any request

Replication factor = 2

nodes are aware of
token ranges

	Row Key Ranges
A	🟡 🔴
B	🟢 🔴
C	🟡 🟣
D	🟢 🟣

C* data model

-
  ```
  CREATE TABLE log (
    sensor_colour   text,
    time_bucket text,  //prevent columns form infinite growth!

    event_id     text,
    timepoint    timestamp,

    properties          text,
    conf         int   // adding a field results in a new column
    PRIMARY KEY ((sensor_colour, time_bucket), event_id, timepoint)
  )
  WITH CLUSTERING ORDER BY (timepoint DESC);
  ```

row_key=sensor_green:11Aug15	col_name=ID1:12131:properties	col_name=ID1:12131:conf
	col_value=on	col_value=6

. . .

CQL

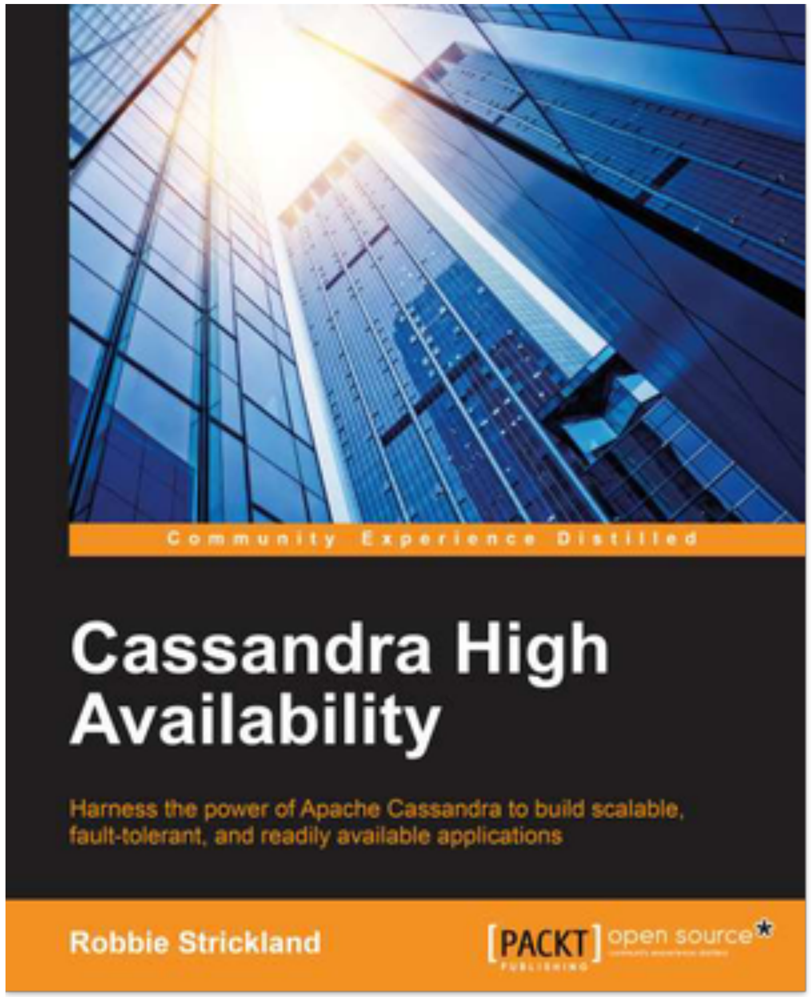
-
  ```
  INSERT (row_key, field_names…) VALUES (?,?,…)
  ```

-
  ```
  SELECT <field_names> FROM <table> WHERE
  row_key OP value AND field_name OP value …
  ```

  -
    ```
    SELECT * FROM log WHERE sensor_colour = “green”
    AND time_bucket = “11Aug2015”
    AND timepoint > 2015-11-83T04:05+0000
    ```
- **No group_by and join operations.**
- Row_key **must be** specified.
  -> Denormalize the data
  -> Build explicit indexes and trees
  -> Build _root_ tables (e.g. a table with one key and a list of all sensors)

Rabbit holes

- Multiple data centers
- Deletion
- Tombstones
- Replication when nodes fail
- Sets, Maps, Lists
- …



Community Experience Distilled

Cassandra High Availability

Harness the power of Apache Cassandra to build scalable, fault-tolerant, and readily available applications

Robbie Strickland

[PACKT] open source*

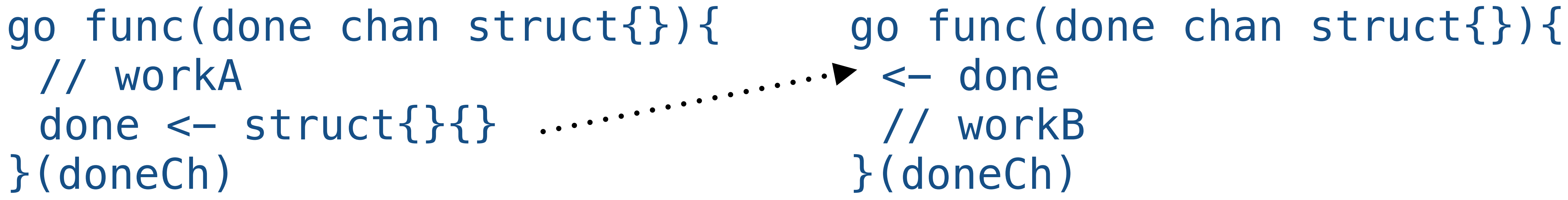
Basic Go concurrency

- (Go) routines are light-weight processes/tasks.

```go
go func(){
  // do work
}()
```

- Channels are basic synchronization primitives. A sender and a receiver block until their traces are in _sync_.

```go
doneCh := make(chan struct{})
```



```go
go func(done chan struct{}){
  // workA
  done <- struct{}{}
}(doneCh)
```

```go
go func(done chan struct{}){
  <- done
  // workB
}(doneCh)
```

workA will always be executed before workB!

Basic Go concurrency

- Channels can be buffered. They become FIFO queues.
  Receivers block if a channel is empty.
  Senders block if a channel is full.

```go
doneCh := make(chan struct{}, 1)
```

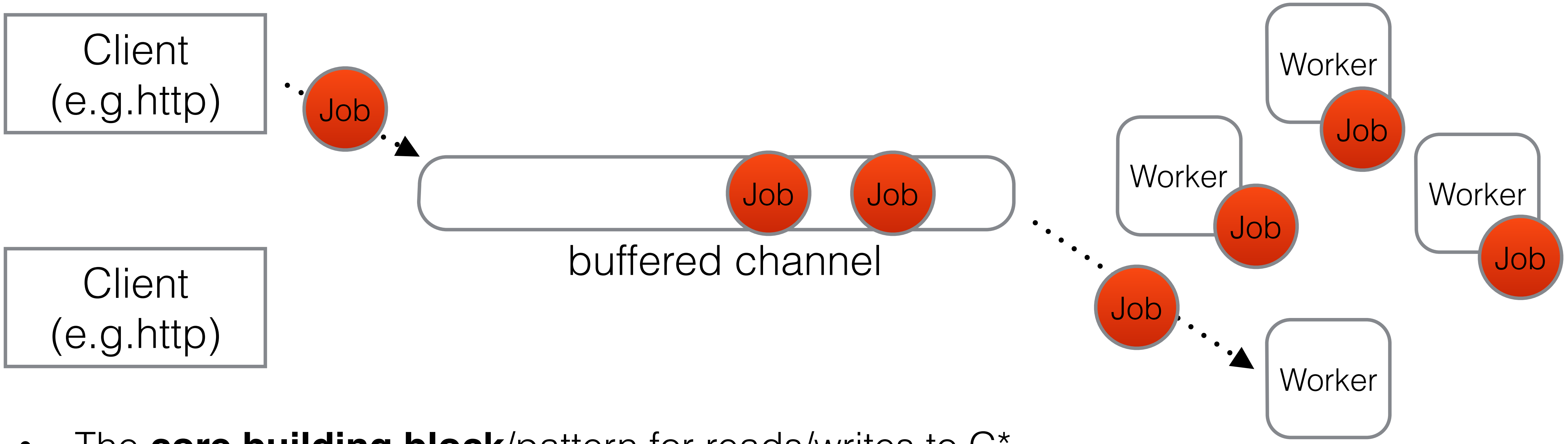
```go
go func(done chan struct{}){
  // workA
  done <- struct{}{}
}(doneCh)
```

```go
go func(done chan struct{}){
  <- done
  // workB
}(doneCh)
```

Now, no guarantees that workA will be executed before workB!

Playtime is over

Simple Worker Pool pattern



- The **core building block**/pattern for reads/writes to C*.
  - Jobs are C* queries.
  - Concurrent routines are Go's mechanism for “non-blocking” IO.
  - One C* node handles multiple reqs, and there are many nodes (set replication).
- **Note.** Though runtime can handle >10^5 routines. Experiment with the # of workers to avoid overwhelming the nodes and thus downgrading the performance.

Simple Worker Pool

```
type Worker struct {
    jobCh <-chan *Job
    dieCh <-chan struct{}
}

func (w Worker) work() {
    for {
        select {
        case <-w.dieCh:
            return

        case job, ok := <-w.jobCh:
            if !ok {
                return
            }

            job.execute()
        }
    }
}
```

Select primitive is a non-deterministic choice between read/writes from/into channels

Loops until the worker is told to die.

Reading from a closed channel sets ok to false.

Simple Worker Pool

```
func RunWorkerPool(nWorkers int, jCh <-chan *Job) chan<- struct{} {
    diePoolCh := make(chan struct{}, 1)
    dieWorkerChs := make([]chan struct{}, nWorkers)

    for i := 0; i < nWorkers; i++ {
        dieCh := make(chan struct{}, 1)
        dieWorkerChs[i] = dieCh

        w := newWorker(jCh, dieCh)
        go w.work()
    }

    go func() {
        <-diePoolCh
        for i := 0; i < nWorkers; i++ {
          dieWorkerChs[i] <- struct{}{}
        }
    }()

    return diePoolCh
}
```

Create a buffered _die_ channel for each worker.

Launch workers and pass _die_ and job channels.

Launch the monitor that kills the workers if the die_pool sig is received.

Give the caller a way to kill the pool.

Rabbit holes

- Automated scaling of workers
  - Monitor the rate of jobs and launch new workers.
  - **Note**, mem_size/2k is a practical limit. Thrashing ensues afterwards :)
- Extending the buffer size to handle spikes
  - Complements launching new workers, if you don’t want uncontrolled growth of routines.
- Possibly add a dispatcher to ease channel contention.
  - In this way workers register for work and can have their queues as well.

Basic insertions with **gocql**

```
type SingleEventInsertJob struct {
  SensorColour string
  TimeBucket string
  EventId  string
  Timepoint time.Time
}
```

Log table mapping

CQL insert query

```
func (j *SingleEventInsertJob) execute() {
  q := “INSERT INTO log(sensor_colour, time_bucket, event_id, timepoint)” +
       "VALUES (?, ?, ?, ?)”;

  query := session.Query(q)
  err := query.Bind(j.SensorColour, j.TimeBucket, j.EventId, j.Timepoint).Exec()
  if err != nil {
    // see later slides
  }

}
```

To **batch** or not

```
type EventListInsertJob struct {
  SensorColour string
  TimeBucket string
  EventIds []string
  Timepoint time.Time
}

func (j *EventListInsertJob) execute() {
  q := “INSERT INTO log(sensor_colour, time_bucket, event_id, timepoint)” +
        "VALUES (?, ?, ?, ?);

  b := session.NewBatch(0)
  for _, e := range j.EventIds {
    b.Query(q, j.SensorColour, j.TimeBucket, j.EventId, j.Timepoint)
  }

  err := session.ExecuteBatch(b)
  if err != nil {
    // see next slide
  }
}
```

Multiple columns!

CAUTION! Only batch when adding columns to the same row_key. Otherwise spawn one job per row_key.

Live **fail-safe** and prosper

```
err := session.ExecuteBatch(b)
if err != nil {
 // Option 1: re-insert the job back into the buffer

 // Option 2: pause the workers with a (non-)linear back-off
              e.g. launch a new routine with a sleep timer.

 // Option 3: dump the _write_ jobs to a local,S3,… storage
              schedule another clean-up process to deal with them.
}
```

- Nodes will fail. There are no 100% availability guarantees in data-centres, especially as clusters grow.

- Network delays will happen (especially in the cloud infrastructure).

- **Fail-safety policy** is a must. It is not “anomalous” behaviour, it should be considered standard.

Basic reads with **gocql**

```
type SimpleSensorReadJob struct {
  SensorColour string
  TimeBucket string
  ResultCh chan interface{}
}

func (j *SimpleSensorReadJob) execute() {
  q := "SELECT event_id FROM log WHERE sensor_colour = ? and time_bucket
  = ?"

  var eventId string

  iter := session.Query(q, j.SensorColour, j.TimeBucket).Iter()
  for {
     if iter.Scan(&eventId) {
         j.ResultCh <- eventId
     } else {
         // signal that there are no more result
     }
  }
  iter.Close() // returns err
}
```

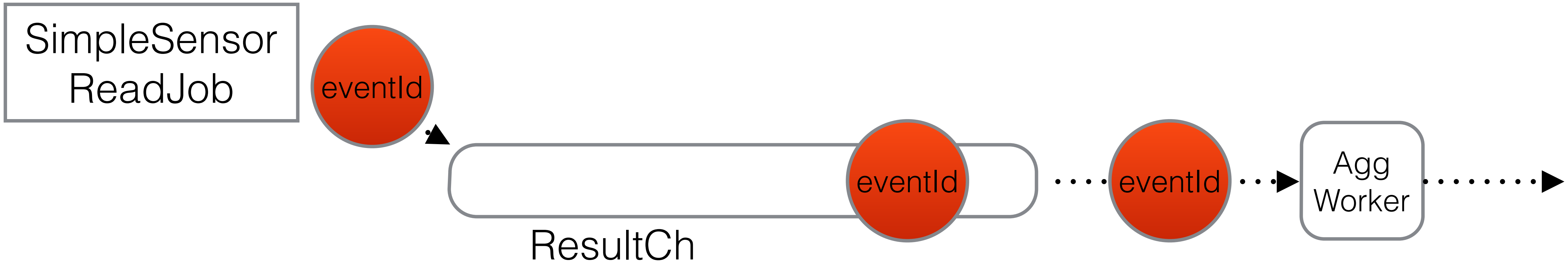
Row key mapping

CQL select query

Decouple scanning from reading!

Don’t forget to close the ResultCh after all jobs are done.

Processing results

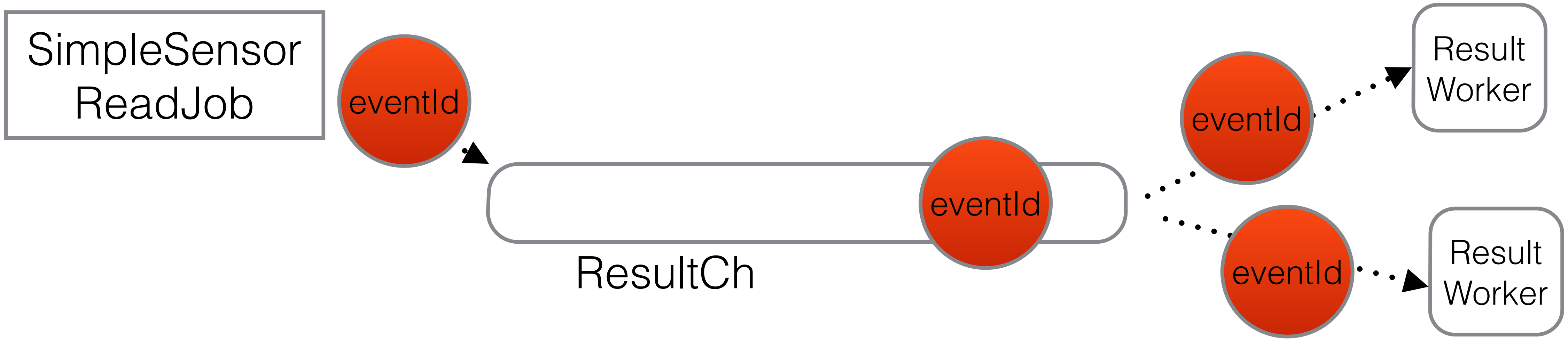


```
type AggWorker struct {
    ResultCh <-chan interface{}
}

func (w AggWorker) work() {
    for {
        select {
        case result, ok := <-w.ResultCh:
            if !ok {
                return
            }

            // do something with the result
        }
    }
}
```

Processing results



```
type CountEventsJob struct {
    EventId int;
    GlobalCounter *int;
    GlobalLock *sync.Mutex;
}

func (j *CountEventsJob) execute() {
    // update counter
}
```

- Reuse the pattern
- Link Worker Pools:
  Group_by, join, aggregator operation are implemented as a sequence of jobs.

Handling complex keys

```
type SensorReadJob struct {
  SensorColour string
  ResultCh chan string
  JobCh chan *Job
}

func (j *SensorReadJob) execute() {
  // Step 1) Compute/fetch the time buckets to create simple jobs.

  // Step 2) Push all simple jobs into a worker pool (via JobCh).

  // Step 3) Process results from ResultCh (as per previous slides).
}
```

- It helps me to think of all reads from Cassandra as Map/Reduce jobs over streams of columns coming from iterators.

- **Note.** DataStax is promoting Spark as *the* framework for reading from C*.

Rabbit holes

- gocql has a number of ways to experiment with the performance
    - Number of connections per node.
    - Number of streams per connection.
    - Number of pages to prefetch.
    - Retry policies.
    - **The code is clean and easy to read and is the place to learn more!**
- Other C* frameworks built on gocql ease with the complex data mapping.
    - gocassa, gocqltable, …

Thank you.

Srdjan Marinovic
@a_little_srdjan

The slides are available at:

https://github.com/a-little-srdjan/cassandra_and_go_presentation