Efficient Field-Striped, Nested, Disk-backed Record Storage

Rob Grzywinski
Aggregate Knowledge, Inc.
San Mateo, CA 94404
rob@aggregateknowledge.com

Requirements

The following is a list of desirable properties of a field-centric storage system:

1. A single field can be sequentially read (in record order) without requiring other fields to be read
2. Losslessly reconstitute the original records including their structure and value order from the individual field stripes
   - Provide for the ability to query for the presence or absence of parent structures
3. Limit the space required by fields that do not have a value
4. No introduction of a synthetic record identifier [say why you do / don't want one of these things]
5. A single pass over the records is necessary to covert the record-oriented data into field-oriented data

Schema

Schemas (see Figure 4 for an example) are defined in a Thrift-meets-Protobuf IDL chosen solely for compactness without sacrificing expressivity. Each field has a unique integer identifier that indicates the field position, an optional reg-ex-like qualifier ("$?$" for zero-or-one, "$*$" for zero-or-more or "$+$" for one-or-more) where no qualifier implies that the field is required, a type, a symbolic name and an optional default value (for primitive types). In multiple-valued fields, order is significant. Schemas cannot contain circular-dependencies – they must be trees rather than graphs. When viewed as a tree, nested types are node fields and primitive types (integer, string, etc) are leaf fields. Schemas are defined *a priori* and are immutable though an extension will be proposed that allow for certain mutations. [Add more goo about trees and how the schema can be viewed as a tree.]

There is no notion of a $NULL$ value. Required fields must have a value otherwise they are invalid. Optional or repeated fields may be unset.

Field-Striped Storage

The challenge is to find a simple and efficient approach to breaking each record into a series of field stripes corresponding to each leaf field in a schema. To simplify exposition, each type of schema will be introduced and examined in detail beginning with a flat schema that contains optional but no repeated fields and proceeding through a nested (hierarchical) structure that contains required, optional and repeated fields. In each case, various encodings will be suggested and any special considerations will be identified.

Encodings follow a natural imperative -- instruction-based -- approach. The rationale is that the encoders and decoders can be best understood as state machine i.e. a set of rules for transitioning between records or field values within a record.

[Is there a better term / phrase than “encoding” that’s less passive and more descriptive?]

Flat Schema

```
struct Employee {
  1:  int64 RecId;
  2:  int64 EmpId;
  3:  int64 DeptId;
  4?: float BonusRate;
  5:  string FirstName;
  6:  string LastName;
}
```

s_1

```
RecId: 1
EmpId: 7342
DeptId: 985
BonusRate: 0.04
FirstName: 'John'
LastName: 'Doe'
```

r_1^1

```
RecId: 2
EmpId: 842
DeptId: 43
FirstName: 'Some'
LastName: 'Guy'
```

r_1^2

Figure 4: Flat schema with example records r_1^1 and r_1^2

A flat schema s_1 with only required and optional (zero-or-one) atomic fields is shown in Figure 4. Notice that record r_1^2 does not have a *BonusRate*.

RecId	EmpId	DeptId	BonusRate	FirstName	LastName
1	7342	985	0.04	John	Doe
2	842	43	*UNSET*	Some	Guy

Figure 5: Field-stripes of example records r_1^1 and r_1^2

Flat schemas require only a trivial striping of the records into individual columns as seen in Figure 5. Missing values for fields are given an *UNSET* value to preserve the original record order. This approach will be used consistently throughout – if there is a parent record (and later, when nested structure are introduced, a parent value) and a child is not specified then an *UNSET* value is used.

This representation is easy to follow in that one reads each column in left-to-right order, row by row. All records for a single field can be read by sequentially reading all values for that field. All records for multiple fields can be read by reading the value from each field in lock-step.

Encoding

The field types are known *a priori* and are, by definition, consistent throughout a single stripe. This provides many advantages when designing the encoding as there does not need to be any per-value token that indicates the type. The only per-value meta-data that is needed is to indicate an *UNSET* value. If desired, additional meta-data can be used to encode multiple sequential *UNSET* values. An example encoding of the *BonusRate* stripe is: *<new-record>*, *0.04*, *<new-record>*, *UNSET*. Because the allowed cardinality of a field is known and fixed *a priori*, it is possible to simplify this encoding to be *0.04*, *UNSET* since it can be assumed that each value implicitly identifies a new record.

Flat Repeated Field Schema

```
struct Employee {                    s2
  1:  int64 RecId;
  2:  int64 EmpId;
  3*: int64 DeptId;
  4?: float BonusRate;
  5:  string FirstName;
  6:  string LastName;
}
```

```
RecId: 1                             r2^1
EmpId: 7342
DeptId: 67
DeptId: 94
BonusRate: 0.04
FirstName: 'John'
LastName: 'Doe'
```

```
RecId: 2                             r2^2
EmpId: 342
FirstName: 'Lou'
LastName: 'Poll'
```

```
RecId: 3                             r2^3
EmpId: 842
DeptId: 43
FirstName: 'Some'
LastName: 'Guy'
```

Figure 6: Repeated field schema s_2 with example records r_2^1, r_2^2 and r_2^3

Figure 6 introduces a slight variation on schema s_1 by making *DeptId* a repeated (zero-or-more) field. Record r_2^1 has multiple *DeptId* field values, r_2^2 has none and r_2^3 has one.

Figure 7 shows the striping of the three records. Notice that due to the multiple values for *DeptId* in r_2^1 one simply cannot reproduce the original records by reading each column left-to-right as was possible with schema s_1. It is not possible to determine which value is associated with which record or which record has a repeated value.

RecId	EmpId	DeptId	BonusRate	FirstName	LastName
1	7342	67	0.04	John	Doe
2	342	94	*UNSET*	Lou	Poll
3	842	*UNSET*	*UNSET*	Some	Guy
		43			

Figure 7: Field-stripes of example records r_2^1, r_2^2 and r_2^3

Encoding

In order to determine which value is associated with which record, repeated values require additional meta-data. This meta-data can take on one of three equivalent forms:

- New record marker: *<new-record>, 67, 94, <new-record>, UNSET, <new-record>, 43*
- Repeated value marker: *67, <repeated-value>, 94, UNSET, 43*
- Repeated value with count marker: *<repeated-value 2>, 67, 94, UNSET, 43*

Repeated fields cannot contain *UNSET* values therefore the first two encodings are not ambiguous with respect to whether or not the first element of the repeated field is *UNSET* versus if the field is not set.

Nested (non-repeated) Field Schema

s_3

```
struct Location {
  1:  string Building;
  2?: int32 Floor;
}
struct Department {
  1:  int64 DeptId;
  2?: string Name;
  3?: Location Loc;
}
struct Employee {
  1:  int64 RecId;
  2:  int64 EmpId;
  3?: Department Dept;
  4?: float BonusRate;
  5:  string FirstName;
  6:  string LastName;
}
```

r_3^1

```
RecId: 1
EmpId: 7342
Dept
  DeptId: 67
  Name: 'Eng'
  Loc
    Building: 'C'
BonusRate: 0.04
FirstName: 'John'
LastName: 'Doe'
```

r_3^2

```
RecId: 2
EmpId: 342
FirstName: 'Lou'
LastName: 'Poll'
```

r_3^3

```
RecId: 3
EmpId: 842
Dept
  DeptId: 43
FirstName: 'Some'
LastName: 'Guy'
```

Figure 8: Nested (non-repeated) field schema s_3 with example records r_3^1, r_3^2 and r_3^3

Schema s_3 shown in Figure 8 introduces optional nested structures. Record r_3^1 has both a department and a location (with only a *Building*), r_3^2 has neither and r_3^3 has only a department (with only a *DeptId*).

RecId	EmpId	BonusRate	FirstName	LastName
1	7342	0.04	John	Doe
2	342	*UNSET*	Lou	Poll
3	842	*UNSET*	Some	Guy

Dept.DeptId	Dept.Name	Dept.Loc.Building	Dept.Loc.Floor
67	Eng	C	*UNSET*
UNSET	*UNSET*	*UNSET*	UNSET
43	*UNSET*	*UNSET*	*UNSET*

Figure 9: Field-stripes of example records r_3^1, r_3^2 and r_3^3

Figure 9 presents the striping of the three records. Similar to schema s_1, which also does not include repeated fields, it is possible to read the records left-to-right. In order to preserve record consistency, each child field must have a value even if a parent value is *UNSET*. This is illustrated in the case of record r_3^2 which has no department (and therefore no location) yet *UNSET* values are required for both *Dept.Loc.Building* and *Dept.Loc.Floor*.

Observe that when only looking at the *Dept.Name* stripe for record r_3^2 it is not possible to determine if there is a department record with an unset name or if there is no department record. Similarly no record contains a *Floor* value but record r_3^1 does contain a *Location*

record. Neither of these meets the requirement of being able to query for the presence or absence of a parent record. An example of such a query is:

```
SELECT * FROM table
WHERE Dept.Location IS NOT NULL
    AND Dept.Location.Floor IS NULL
```

The result of this query is expected to be r_3^1. Continuing one step further, it is not sufficient to only know if an immediate-parent record is *UNSET* as demonstrated in the following query:

```
SELECT * FROM table
WHERE Dept IS NOT NULL
    AND Dept.Location.Floor IS NULL
```

whose expected results are r_3^1 and r_3^3. In an *n*-level nested structure, it is necessary to know if any of the *n-1* parents are *UNSET*. Because it is not possible to have an *UNSET* parent of a non- *UNSET* child, only the first *UNSET* parent (the farthest from the child leaf) needs to be known. (An *UNSET* parent implies an *UNSET* child.)

Encoding

Dept
SET
SET
UNSET

Dept.Loc
SET
UNSET
UNSET

Figure 10: Node field-stripes of example records r_3^1, r_3^2 and r_3^3

In order to determine if a parent record exists for an optional field, additional meta-data is required. This meta-data can take on two forms:

1. Up to this point only leaf fields have been striped. It is possible to stripe and encode node fields. Refer to Figure 10 for an example. This approach is not desirable as it is necessary in the worst-case to walk *n-1* parent node stripes.
2. An explicit *parent-record-is-UNSET* marker that indicates the depth (from the root) of the first *UNSET* parent. Note that this marker would take the place of an *UNSET* value marker since it is not possible to have a non-*UNSET* child of an *UNSET* (parent) value.

An example encoding of *Dept.Loc.Floor* is *UNSET*, *<parent-is-UNSET 1>*, *<parent-is-UNSET 2>* (using the fact that *Dept* is at a depth of 1 and *Dept.Loc* is at depth of 2) where it is assumed that each value represents a new record.

Observe that required parents cannot have an *UNSET* value. This allows an optimization to be made where the parent depth does not need to count required parents.

Nested Repeated Field Schema

s_4

```
struct Location {
  1:  string Building;
  2*: int32 Floor;
}
struct Department {
  1:  int64 DeptId;
  2?: string Name;
  3*: Location Loc;
}
struct Employee {
  1:  int64 RecId;
  2:  int64 EmpId;
  3*: Department Dept;
  4?: float BonusRate;
  5:  string FirstName;
  6:  string LastName;
}
```

r_4^1

```
RecId: 1
EmpId: 7342
Dept
  DeptId: 67
  Name: 'Eng'
  Loc
    Building: 'C'
  Loc
    Building: 'D'
Dept
  DeptId: 94
  Loc
    Building: 'C'
    Floor: 2
    Floor: 3
BonusRate: 0.04
FirstName: 'John'
LastName: 'Doe'
```

r_4^2

```
RecId: 2
EmpId: 342
FirstName: 'Lou'
LastName: 'Poll'
```

r_4^3

```
RecId: 3
EmpId: 842
Dept
  DeptId: 43
FirstName: 'Some'
LastName: 'Guy'
```

Figure 11: Nested repeated field schema s_4 with example records r_4^1, r_4^2 and r_4^3

Figure 11 shows schema s_3 extended to include repeated fields. The repetition takes on two distinctly different forms: repeated node fields (e.g. *Dept* and *Loc*) and repeated leaf fields (e.g. *Dept.Name* and *Dept.Loc.Floor*). Record r_4^1 has two departments one of which is located on two floors of one building and the other is in two buildings, r_4^2 has neither and r_4^3 has only a department (with only a *DeptId*).

Dept.DeptId
67
94
UNSET
43

Dept.Name
Eng
UNSET
UNSET
UNSET

Dept.Loc.Building
C
D
C
UNSET
UNSET

Dept.Loc.Floor
UNSET
UNSET
2
3
UNSET
UNSET

Figure 12: Nested field-stripes of example records r_4^1, r_4^2 and r_4^3

The non-nested fields have been omitted for brevity from Figure 12 (they are found in Figure 9). Striping of the *Dept.Name*, *Dept.Loc.Building* and *Dept.Loc.Floor* fields present special challenges beyond what has been described in the nested non-repeated fields section. If a parent field exists then all child field values must be represented even if those values are *UNSET*. This can be best seen in the *Dept.Loc.Floor* stripe. Because there are two department values in record r_4^1, there must be ***at least*** two values in the *Dept.Loc.Floor* stripe even if there

were no explicit location values. Each of these *Dept.Loc.Floor* values needs to be associated with the appropriate parent. To do this the following three states must be disambiguated:

1. New record
2. New repeated parent value (where there may be multiple repeated parents)
3. New repeated value

The distinction between a new parent and a repeated parent in the *repeated-parent* case must be made. The first *UNSET* in *Dept.Loc.Floor* is associated with a new parent, specifically the parent of the shallowest depth – a new record. The second *UNSET* is associated with the immediate repeated parent *Dept.Loc* (depth of 2). The value *2* must be associated with the repeated parent *Dept* (depth of 1) rather than *Dept.Loc* since that would be ambiguous with the previous case. The value *3* is simply a case of *repeated-value* following from the repeated field section. The third *UNSET* occurs in a new record and with an *UNSET Dept* (depth of 1) – no other parent has been repeated. The fourth and final *UNSET* occurs in a new record with an *UNSET Dept.Loc* (depth of 2).

Encoding

An encoding of *Dept.Loc.Floor* is *UNSET, <repeated-parent 2>, UNSET, <repeated-parent 1>, 2, <repeated-value>, 3, <parent-is-UNSET 1>, <parent-is-UNSET 2>. UNSET* and *parent-is-UNSET* markers always indicate a new record unless either the *repeated-value* or *repeated-parent* markers precede them. Notice that both the *parent-is-UNSET* and *repeated-parent* meta-data are required to provide for a lossless encoding.

An encoding of *Dept.Loc.Building* is *'C', <repeated-parent 2>, 'D', <repeated-parent 1>, 'C', <parent-is-UNSET 1>, <parent-is-UNSET 2>*.

Observe that only repeated parents need to be counted in the parent depth for a *repeated-parent* marker while both optional and repeated parents need to be counted in the parent depth for a *parent-is-UNSET* marker.

Encoding

From the above cases its possible to extract the following set of instructions:

- *value <value>*
- *UNSET*
- *parent-is UNSET <optional and repeated parent depth>*
- *repeated-value*
- *repeated-parent <repeated parent depth>*

These can be categorized into two groups: *value*, *UNSET* and *parent-is-UNSET* are value-based instructions and *repeated-value* and *repeated-parent* are modifier-based instructions. A value-based instruction indicates a new record unless preceded by a modifier-based instruction.

The above instruction set can be used generically in all cases but for a more efficient encoding the following cases can be followed:

- Required field with either no or only required parents: values (with no meta-data) can simply be encoded sequentially.
- Required field: *value* instruction.
- Optional field: *value* and *UNSET* instructions.
- Repeated field: *value*, *UNSET* and *repeated-value* instructions.
- At last one optional parent: include the *parent-is-UNSET* instruction.
- At least one repeated parent: include the *parent-is-UNSET* and *repeated-parent* instructions.

For example, for an optional field with one optional parent and one repeated parent the required instructions are *value* and *UNSET* for the optional field, *parent-is-UNSET* for the optional parent (as well as the repeated parent) and *repeated-parent* for the repeated parent.

Algorithm Construction

A tree of encoders is created to match the fields is the schema. Records are both parsed and encoded by following a depth-first traversal of the tree. Node encoders exist simply to read any unset or repeated node fields and propagate that state information to their children. For example, if a node field is unset then that information (along with the depth of the unset node) is propagated to each child and eventually written by the leaf encoders. The field encoders either read and encode their field's value or write the meta-data that was propagated from a parent to the field-stripes.

Decoding

Decoding comes in two flavors: record reassembly and query [is this the best name?].

Record Reassembly

Tree-based Reassembly

The goal of record reassembly is to losslessly reassemble the original records from the field stripes. This can be accomplished intuitively with the construction of a tree of decoders that matches the schema. As with encoding, a depth-first traversal of the tree is performed. (A breadth-first approach is not possible since only the leaf-nodes are associated with field-stripes. Specifically, a node has no information without first asking its children which implies depth-first.) Node decoders write out any structure information associated with their field or write *UNSET* if their depth is the depth of the *parent-is-UNSET* meta-data. Leaf decoders parse their field-stripe and either propagates any *parent-is-UNSET* or *repeated-parent* meta-data up to their parents or writes out their values.

The only challenge in decoding using a tree-based approach stems from the fact that node decoders receive all meta-data from their children leaf decoders. Based on the output format of the assembled records (e.g. JSON, XML) it may be necessary for a node to have its children look ahead to determine if its value is set and only if it is then write out any required structure information (e.g. braces for JSON or an open-tag for XML). Since all of the children have the same *repeated-parent* and *parent-is-UNSET* meta-data, only one child needs to perform the look ahead.

Finite-State Machine-based Reassembly

While the tree-based approach may be the most intuitive it is not the most performant. Since only the leaf-fields are associated with a field-stripe, it would seem that a more performant algorithm would only involve those leaf fields. A finite-state machine (FSM) in which each field is a state and a new record or a repeated value/parent is a transition can be employed.

Because only leaf fields are represented in field stripes transitions are made *to* the first leaf field for a parent node. Transitions are made *from* the last field node of a repeated parent back to its first child leaf field. An example of a state machine is shown in Figure 13. Next record transitions are marked with *NR*, *repeated-value* transitions are marked with *RV* and *repeated-parent* transitions are marked with the depth of the repeated parent.

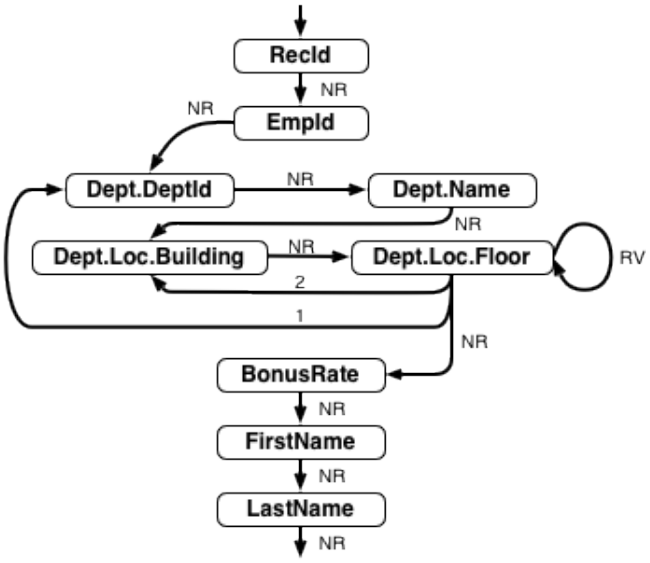

Figure 13: Finite-state machine for record reassembly of schema s_4

Some transitions indicate that a structure change has occurred and must be reflected in the reassembled record. For example, in the transition from *EmpId* to *Dept.DeptId* the structure changes from *root* to *root.Dept* which implies that the *Dept* structure must be started in the reassembled record.

The following presents a walk through of the state machine in Figure 13:

- Record reassembly begins by transitioning to the *RecId* state. In this transition the parent changes from nothing to *root* which is recorded in the reassembled record as a start-of-record. The first value is read and written to the reassembled record. Since *RecId* is a required field with no parents, only values are stored in the field-stripe and each value represents a new record.
- A transition is made to the *EmpId* state. There is no parent change between these two states. The first value is read and written to the reassembled record.
- A transition is made to the *Dept.DeptId* field – the first leaf field of the repeated *Dept* node field. *Dept.DeptId* is a required field with a repeated parent which allows for the *value*, *parent-is-UNSET* and *repeated-parent* instructions. The parent hierarchy changed from *root* to *root.Dept*. If *parent-is UNSET* is encountered (with a depth of *1* as that is the only allowed depth) then *UNSET* is recorded in the reassembled record since that parent changed. If *repeated-parent* is encountered then nothing is done until transitioning away from the *Dept* parent. If *value* is encountered then the start of the *Dept* structure is written to the reassembled record (since the parent changed) and the read value is appended.
- A transition is made to the *Dept.Name* state which is an optional field with a repeated parent (supporting the *value*, *UNSET*, *parent-is-UNSET* and *repeated-parent* instructions). The parent hierarchy did not change. If *parent-is-UNSET* or *repeated-parent* is read then nothing is reflected in the reassembled record since it would have been accounted for

in the previous field (when the parent hierarchy changed). *value* or *UNSET* cause either the value or an *UNSET* to be recorded in the reassembled record.

- A transition is made to *Dept.Loc.Building* which is a required field with a repeated parent (supporting the *value*, *parent-is-UNSET* and *repeated-parent* instructions). The *Loc* parent has changed. If *parent-is-UNSET* is encountered with a depth of *2* then *UNSET* is recorded in the reassembled record but if a depth of *1* is encountered then nothing is done as it was handled when the *Dept* parent changed in *Dept.DeptId*. *repeated-parent* causes no change whereas *value* causes the start of the Loc structure to be written to the reassembled record followed by the read value.
- A transition is made to *Dept.Loc.Floor* which is a repeated field with repeated parents (supporting all possible instructions). The parent hierarchy did not change. If *parent-is-UNSET* regardless of depth or *repeated-parent* with a depth of is read then nothing is reflected in the reassembled record (since the parent hierarchy did not change). If *value* is read then that value is written to the reassembled record.
- Beside the expected new-record transition there are three other possible transitions:
    - *repeated-value* transitions back to itself;
    - *repeated-parent* with a depth of *2* transitions back to the first leaf node of the parent *Dept.Loc* which is *Dept.Loc.Building*. Because the parent hierarchy does not change in this transition no addition information is written to the reassembled record;
    - *repeated-parent* with a depth of *1* transitions back to the first leaf node of the parent *Dept* which is *Dept.DeptId*. Because the parent hierarchy changes from *root.Dept.Loc* to *root.Dept* the *Loc* parent structure must be ended in the reassembled record;
- A transition is made to BonusRate which is an optional field with no parents. The parent hierarchy changes from *root.Dept.Loc* to *root* requiring that both the *Loc* and *Dept* parent structures must be ended in the reassembled record. Either the value or *UNSET* are appended to the reassembled record.
- A transition is made to the *FirstName* state. The parent hierarchy did not change and the read value is appended to the reassembled record.
- A transition is made to the *LastName* state. The parent hierarchy did not change and the read value is appended to the reassembled record.
- No states remain and the reassembled record is finished.

Partial Field Reassembly

Both approaches to decoding can be extended to using only a subset of the field stripes to reassemble what the records would have been had they only included that subset of fields. In the tree-based approach, a tree that only contains the desired leaf fields and their parent nodes is created. In the FSM-based approach, states and transitions to those states are created only for the desired fields. For example to reassemble records that only include the *EmpId*, *Dept.DeptId* and *Dept.Loc.Floor* from schema s_4 the state machine shown in Figure 14 is used.

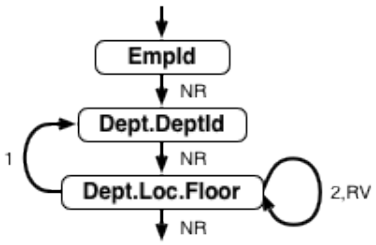

Figure 14: Finite-state machine for reassembly of a subset of fields from schema s_4

[Cover the odd '2' case in Figure 14]

Breadth-first "Trickle" Reassembly

The final approach to reassembly involves advancing the decoder of each field-stripe only in the case where the *repeated-parent* depth is not less than a certain value. If there are no more repeated values then all decoders are advanced to the next record. This causes the record to be reassembled in a breadth-first manner that "trickles" the data into each field of the record as repeated values are read. An example using schema s_4 is shown in Figure 15.

RecId	EmpId	DeptId	Name	Building	Floor	BonusRate	FirstName	LastName
1	7342	67	Eng	C	*UNSET*	0.04	John	Doe
.	.	.	.	D	*UNSET*	.	.	.
.	.	94	*UNSET*	C	2	.	.	.
.	.	.	.	.	3	.	.	.
2	342	*UNSET*	*UNSET*	*UNSET*	*UNSET*	*UNSET*	Lou	Poll
3	842	43	*UNSET*	*UNSET*	*UNSET*	*UNSET*	Some	Guy

Figure 15: "Trickle" record reassembly of example records r_4^1, r_4^2 and r_4^3
(field names have been shortened for space reasons)

[Something about converting breadth-first to depth-first to allow for actual record reassembly.]
[Pre-mention something about why one would do this?]

Query

This will focus on restriction (selection) and projection. No standard exist per se that define how to perform restriction against hierarchical, repeated data. [The closest would be XQuery.]

Artificially limit the implementation to the following requirements:

1. A single pass of the field-striped data. Implies that restriction and projection must be performed as the data is read rather than in two separate steps.
2. SAX-style reader (i.e. the record is not cached) where at most a single value is known for all fields / readers [phrase better!]

What is the most that can be achieved in this configuration?

- Arbitrary restrictions that result in record pruning are not possible. Given a restriction and projection on field A and a restriction and projection on field B which is in a different sub-tree from field A, it is possible to construct a record that will project field A but then learn that it should have been pruned due to the restriction on field B.
- At best sub-tree pruning is allowed. As traversal of the field-stripes occurs in a breadth-first manner, restrictions prune all fields with a depth greater than that of the restricted field (for a given sub-tree). Projections occur only after all restrictions have been performed at a given depth [rephrase]. [Doesn't work with $Dept.DeptId=yadda$ and $Dept.Loc=yadda$ since $Dept.DeptId$ would be projected before all values of $Dept.Loc$ are restricted!]

[finish]

Conclusion

[Finish!]

[talk about compression]

[talk about actual writing with multiple writers]

[note that all optional means is “can have null”]

[optional is really “null”able (rather than “doesn’t exist”)]

[move some of the editorial to the conclusion]

Ideally, each stripe would be placed onto its own spindle to minimize head-seek time when cycling between fields. In practice though, this may not be feasible. Taking from common NoSQL practices, multiple stripes can be combined into a stripe-family (akin to a “column family”) in which often parallel accessed columns are interleaved. If multiple stripes are located on a single spindle then fully utilizing buffering and tuning it based on the head-seek time should provide an adequate result.

Intro notes:

[main points:

- Motivate why column-based makes sense (only need to read the part of the data that one is interested in – only read what you want to read)
- Introduce hierarchical data
- Why isn’t this “HBase”?
    - This is an encoding of data not a NoSQL solution

]

One of the fundamental tenants of Map-Reduce is to push the computation to the data. The standard approach taken in most map implementations is to iterate over the input data set record by record and allowing the user-defined mapper to filter the fields and records to pass to the reducer. When using record-centric storage, all of the data must be touched even though only a small subset of either the records or fields are desired.

[insert pic of record-based storage and iteration]

One approach to reduce the amount of data read is to divide the records into their individual fields and store each field in their own *stripe*. This approach is known as columnar storage from the RDBMs world of tables, rows and columns. As long as the fields (“columns”) can be matched records (“rows”) can be reconstructed in whole or in part then only those fields that are needed for the mapper need to be read.

[insert pic of column-based storage and iteration]

As we strive to better model our domains it becomes necessary to introduce nested (hierarchical) structures into the models. A column-based encoding of the nested data is required. [Finish]

[insert pic of a nested-structured record]

[Define stripe]