PHOENIX-374: Enable access to dynamic columns in * or cf.* selection

Problem Statement and Approaches:

Currently, wildcard queries do not contain the dynamic column data unless the user specifically provides the name and data type of the dynamic columns as outlined in the example here - https://phoenix.apache.org/dynamic_columns.html. It is extremely tedious for the end-user to keep track of dynamic columns for each row, especially since each row may have any number of dynamic columns. In order to support this, we need to broadly achieve the following:

1. **Store metadata pertaining to dynamic columns:** We need to be able to store the metadata for dynamic columns so that wildcard queries are able to use this metadata to get the data type, scale, etc. information which is generally passed in as part of the Expression metadata for each column. Approaches:
    - **Use SYSTEM.CATALOG:** This approach is not scalable since each row may have thousands of dynamic columns.
    - **Cell tagging:** Tag each cell corresponding to a dynamic column with the serialized protobuf representation of the PColumn. This approach was later rejected since cell tags are encoded and encoding-decoding + extracting relevant tags for each cell would be expensive. In addition to this, cell tags are generally not meant for storing large amounts of user-defined metadata, but instead are used for storing smaller metadata such as ACL, TTL (generally a couple of bytes).
    - **Store metadata in reserved(shadow) cells:** We would use 1 cell per dynamic column per row to store the metadata for each respective dynamic column. We use reserved qualifier for these cells.
2. **Resolve dynamic column data using metadata:** We will use the metadata for each dynamic column to resolve data for dynamic columns in case of wildcard queries.
3. **Expose dynamic columns in the PhoenixResultSet:** For the end-user to access dynamic column data, we need to provide necessary APIs in the PhoenixResultSet.

Limitations:

Currently, dynamic columns are not supported for immutable tables with IMMUTABLE_STORAGE_SCHEME = SINGLE_CELL_ARRAY_WITH_OFFSETS storage scheme. **See PHOENIX-5107**, hence we will only be supporting the ONE_CELL_PER_COLUMN storage scheme, however since metadata is stored in separate cells, SINGLE_CELL_ARRAY_WITH_OFFSETS should also work once PHOENIX-5107 is fixed.

Implementation:

Upsert Dynamic Column Data Path:

1. Make this configurable based on a property ("phoenix.query.wildcard.dynamicColumns") whose default value is false.
2. Define our own DynamicColumnMetaData.proto which has a repeated field of PColumn to store dynamic column metadata.
3. We store a mapping from column family to list of dynamic columns inside the PRowImpl.
4. Based on the above-mentioned mapping, we set attributes on our Put mutations with key as the <column family name> and value as the serialized list of PColumns for dynamic columns. This is done inside MutationState#generateMutations().
5. In case the configuration is on and there are dynamic columns for which we need to store metadata, we set an additional attribute on the Put to indicate this.
6. In case of 1) old clients, 2) clients for whom this configuration is off, 3) clients for whom this configuration is on and there are no dynamic columns to process in the mutation, this additional attribute will not be set. Thus, if this attribute is not set, we can avoid unnecessary iterations over each put's column families in the preBatchMutate hook.
7. Add a preBatchMutate hook to the ScanRegionObserver which intercepts the batch mutations and performs an additional Put operation using addOperationsFromCP so that we can add cells for the metadata for each dynamic column.
8. Use 1 shadow cell per dynamic column per row to store dynamic column metadata for each dynamic column in the row. We will use some reserved qualifier ("D#<column qualifier of the dynamic column>") for this and the value of the cell will be the serialized protobuf PColumn of the dynamic column.
9. Since the shadow cells' qualifiers contain '#' (a reserved character), no extra logic needs to be added to prevent users from creating columns that interfere with them.

Select (Table/Column Family) Wildcard Path:

1. For wildcard queries and when the config is on, do not push any columns into the scan i.e. in ProjectionCompiler.compile, do not projectAllColumnFamilies on the scan. In case of column family wildcard queries, we return a RowProjector with isProjectDynColsInWildcardQueries set, which we check on the client-side later.
2. Set a specific scan attribute to indicate that we need to process the dynamic column metadata cells "_WildcardScanIncludesDynCols".
3. When we get the iterators for the scan we need to make sure that we don't push the FirstKeyOnlyFilter filter into the scan (current code thinks that if the scan has no columns projected in it, we only want the row key), so do not do this if our config is true.

4. Note that currently, we serialize the TupleProjector for static columns in our scan using the “scanProjector” attribute. We currently use this projector to project static column data.
5. When we issue an rs.next(), on the server-side i.e. in RegionScannerFactory#getWrappedScanner, we need to ensure that we iterate over the list of cells returned from the scan and use the dynamic column metadata (PColumns for each dynamic column) to figure out which cells correspond to dynamic columns (we can do this since we know that the metadata qualifier is D#_<actual dynamic column qualifier>).
6. We then use the list of dynamic column PColumns and cells corresponding to dynamic columns to create the TupleProjector for dynamic columns, as well as merge the projected values with the projected values of static columns (using TupleProjector#mergeProjectedValue).
7. Finally, we append the serialized list of PColumns for dynamic columns to the end of the tuple (prefixed with known bytes of ‘D#_’ for easy parsing on the client-side), ultimately replacing our List<Cell> result with a single byte[] containing <Actual projected value bytes><D#_ bytes><byte array corresponding to the serialized list of dynamic column PColumns> (see TupleProjector#mergeWithDynColsListBytesAndGetValue)
8. On the client-side, when we get the currentRow inside PhoenixResultSet, we need to parse the byte[] since it originally contained just the <Actual projected value bytes>, so we parse out the byte[] corresponding to the list of dynamic column PColumns, deserialize it and use this to create ProjectedColumnExpressions for dynamic columns.
9. We finally store the combined row projector within the PhoenixResultSet as rowProjectorWithDynamicCols and use this instead of the regular rowProjector in case the config is on and we have a wildcard query.
10. Note that when initializing the PhoenixResultSet object, if the config is enabled, we also figure out the starting position for dynamic columns by processing the known columns. We store the list of “static” columns and use this when creating the combined RowProjector for known (static) + dynamic columns.

How to Expose Dynamic Column Values in PhoenixResultSet:

1. Since we store the combined row projector and use it as required, we can simply use PhoenixResultSetMetadata#getColumnCount() to return the total number of columns (regular + dynamic colums) and use getColumnType(), getColumnName() metadata APIs. We can get the actual data using PhoenixResultSet#getObject().

Future Improvements:

See PHOENIX-5129 wherein we discuss the idea of storing the metadata for dynamic columns in the same cell as the data for the dynamic cells rather than using shadow cells, in order to reduce the storage required for store files.